ISO/IEC JTC 1/SC 22 N 4411

2008-10-09

ISO/IEC JTC 1/SC 22
Programming Languages

Document Type:	**Text for CD Ballot**
Document Title:	**ISO/IEC 14882, Programming language – C++**
Document Source:	**WG 21 Convener (P.J. Plauger)**
Document Status:	**This document is circulated to National Bodies for ballot. Please submit your vote via the online balloting system by the due date indicated.**
Action ID:	
Due Date:	**200**9-01-10
No. of Pages:	**1314**

Secretariat, ISO/IEC JTC 1, American National Standards Institute, 25 West 43rd Street, New York, NY 10036; Telephone: 1 212 642 4932; Facsimile: 1 212 840 2298; Email: lrajchel@ansi.org

ISO/IEC JTC 1/SC22/WG21 N 2800

Date: 2008-10-08

ISO/IEC IS 14882

ISO/IEC JTC 1/SC22

Secretariat: ANSI

Programming Languages — C++

Langages de programmation — C++

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Draft International Standard
Document stage: (30) Committee
Document Language: E

Contents

List of Tables

1 General [intro]

1.1 Scope [intro.scope]

1 This International Standard specifies requirements for implementations of the C++ programming language. The first such requirement is that they implement the language, and so this International Standard also defines C++. Other requirements and relaxations of the first requirement appear at various places within this International Standard.

2 C++ is a general purpose programming language based on the C programming language as described in ISO/IEC 9899:1990 *Programming languages — C* (1.2). In addition to the facilities provided by C, C ++ provides additional data types, classes, templates, exceptions, namespaces, inline functions, operator overloading, function name overloading, references, free store management operators, and additional library facilities.

1.2 Normative references [intro.refs]

1 The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

- Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.
- ISO/IEC 2382 (all parts), *Information technology — Vocabulary*
- ISO/IEC 9899:1990, *Programming languages — C*
- ISO/IEC 9899/Amd.1:1995, *Programming languages — C, AMENDMENT 1: C Integrity*
- ISO/IEC 9899:1999, *Programming languages — C*
- ISO/IEC 9899:1999/Cor.1:2001, *Programming languages — C*
- ISO/IEC 9899:1999/Cor.2:2004, *Programming languages — C*
- ISO/IEC 9945:2003, *Information Technology — Portable Operating System Interface (POSIX)*
- ISO/IEC TR 10176:2003, *Information technology — Guidelines for the preparation of programming language standards*
- ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*
- ISO/IEC TR 19769:2004, *Information technology — Programming languages, their environments and system software interfaces — Extensions for the progrmming language C to support new character data types*

2 The library described in Clause 7 of ISO/IEC 9899:1990 and Clause 7 of ISO/IEC 9899/Amd.1:1995 is hereinafter called the *C standard library*.[1]

1) With the qualifications noted in Clauses 17 through 27, and in C.2, the C standard library is a subset of the C++ standard library.

3 The library described in Clause 7 of ISO/IEC 9899:1999 and Clause 7 of ISO/IEC 9899:1999/Cor.1:2001 and Clause 7 of ISO/IEC 9899:1999/Cor.2:2003 is hereinafter called the *C99 standard library*.

4 The library described in ISO/IEC TR 19769:2004 is hereinafter called the *C Unicode TR*.

5 The operating system interface described in ISO/IEC 9945:2003 is hereinafter called *POSIX*.

6 The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called *ECMA-262*.

1.3 Definitions [intro.defs]

1 For the purposes of this International Standard, the definitions given in ISO/IEC 2382 and the following definitions apply. 17.3 defines additional terms that are used only in Clauses 17 through 27 and Annex D.

2 Terms that are used only in a small portion of this International Standard are defined where they are used and italicized where they are defined.

1.3.1 [defns.argument]

argument

an expression in the comma-separated list bounded by the parentheses in a function call expression; a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation; the operand of `throw`; or an expression, *type-id* or *template-name* in the comma-separated list bounded by the angle brackets in a template instantiation. Also known as an *actual argument* or *actual parameter*.

1.3.2 [defns.cond.supp]

conditionally-supported

a program construct that an implementation is not required to support. [*Note:* Each implementation documents all conditionally-supported constructs that it does not support. — *end note*]

1.3.3 [defns.diagnostic]

diagnostic message

a message belonging to an implementation-defined subset of the implementation's output messages.

1.3.4 [defns.dynamic.type]

dynamic type

the type of the most derived object (1.8) to which the lvalue denoted by an lvalue expression refers. [*Example:* if a pointer (8.3.1) `p` whose static type is "pointer to class `B`" is pointing to an object of class `D`, derived from `B` (Clause 10), the dynamic type of the expression `*p` is "`D`." References (8.3.2) are treated similarly. — *end example*] The dynamic type of an rvalue expression is its static type.

1.3.5 [defns.ill.formed]

ill-formed program

input to a C++ implementation that is not a well-formed program.

1.3.6 [defns.impl.defined]

implementation-defined behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation documents.

1.3.7 **[defns.impl.limits]**

implementation limits

restrictions imposed upon programs by the implementation.

1.3.8 **[defns.locale.specific]**

locale-specific behavior

behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

1.3.9 **[defns.multibyte]**

multibyte character

a sequence of one or more bytes representing a member of the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set (2.2).

1.3.10 **[defns.parameter]**

parameter

an object or reference declared as part of a function declaration or definition, or in the catch Clause of an exception handler, that acquires a value on entry to the function or handler; an identifier from the comma-separated list bounded by the parentheses immediately following the macro name in a function-like macro definition; or a *template-parameter*. Parameters are also known as *formal arguments* or *formal parameters*.

1.3.11 **[defns.signature]**

signature

the name and the parameter-type-list (8.3.5) of a function, as well as the class, concept, concept map, or namespace of which it is a member. If a function or function template is a class member its signature additionally includes the *cv*-qualifiers (if any) and the *ref-qualifier* (if any) on the function or function template itself. The signature of a constrained member (9.2) includes its template requirements. The signature of a function template additionally includes its return type, its template parameter list, and its template requirements (if any). The signature of a function template specialization includes the signature of the template of which it is a specialization and its template arguments (whether explicitly specified or deduced). [*Note:* Signatures are used as a basis for name mangling and linking. — *end note*]

1.3.12 **[defns.static.type]**

static type

the type of an expression (3.9), which type results from analysis of the program without considering execution semantics. The static type of an expression depends only on the form of the program in which the expression appears, and does not change while the program is executing.

1.3.13 **[defns.undefined]**

undefined behavior

behavior, such as might arise upon use of an erroneous program construct or erroneous data, for which this International Standard imposes no requirements. Undefined behavior may also be expected when this International Standard omits the description of any explicit definition of behavior. [*Note:* permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving

during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message). Many erroneous program constructs do not engender undefined behavior; they are required to be diagnosed. — *end note*]

1.3.14 **[defns.unspecified]**

unspecified behavior

behavior, for a well-formed program construct and correct data, that depends on the implementation. The implementation is not required to document which behavior occurs. [*Note:* usually, the range of possible behaviors is delineated by this International Standard. — *end note*]

1.3.15 **[defns.well.formed]**

well-formed program

a C++ program constructed according to the syntax rules, diagnosable semantic rules, and the One Definition Rule (3.2).

1.4 Implementation compliance [intro.compliance]

1 The set of *diagnosable rules* consists of all syntactic and semantic rules in this International Standard except for those rules containing an explicit notation that "no diagnostic is required" or which are described as resulting in "undefined behavior."

2 Although this International Standard states only requirements on C++ implementations, those requirements are often easier to understand if they are phrased as requirements on programs, parts of programs, or execution of programs. Such requirements have the following meaning:

- If a program contains no violations of the rules in this International Standard, a conforming implementation shall, within its resource limits, accept and correctly execute[2] that program.
- If a program contains a violation of any diagnosable rule or an occurrence of a construct described in this Standard as "conditionally-supported" when the implementation does not support that construct, a conforming implementation shall issue at least one diagnostic message.
- If a program contains a violation of a rule for which no diagnostic is required, this International Standard places no requirement on implementations with respect to that program.

3 For classes and class templates, the library Clauses specify partial definitions. Private members (Clause 11) are not specified, but each implementation shall supply them to complete the definitions according to the description in the library Clauses.

4 For functions, function templates, objects, and values, the library Clauses specify declarations. Implementations shall supply definitions consistent with the descriptions in the library Clauses.

5 The names defined in the library have namespace scope (7.3). A C++ translation unit (2.1) obtains access to these names by including the appropriate standard library header (16.2).

6 The templates, classes, functions, and objects in the library have external linkage (3.5). The implementation provides definitions for standard library entities, as necessary, while combining translation units to form a complete C++ program (2.1).

7 Two kinds of implementations are defined: *hosted* and *freestanding*. For a hosted implementation, this International Standard defines the set of available libraries. A freestanding implementation is one in which

2) "Correct execution" can include undefined behavior, depending on the data being processed; see 1.3 and 1.9.

execution may take place without the benefit of an operating system, and has an implementation-defined set of libraries that includes certain language-support libraries (17.6.2.4).

8 A conforming implementation may have extensions (including additional library functions), provided they do not alter the behavior of any well-formed program. Implementations are required to diagnose programs that use such extensions that are ill-formed according to this International Standard. Having done so, however, they can compile and execute such programs.

9 Each implementation shall include documentation that identifies all conditionally-supported constructs that it does not support and defines all locale-specific characteristics.[3]

1.5 Structure of this International Standard [intro.structure]

1 Clauses 2 through 16 describe the C++ programming language. That description includes detailed syntactic specifications in a form described in 1.6. For convenience, Annex A repeats all such syntactic specifications.

2 Clauses 18 through 30 and Annex D (the *library Clauses*) describe the Standard C++ library, which provides definitions for the following kinds of entities: macros (16.3), values (Clause 3), types (8.1, 8.3), templates (Clause 14), classes (Clause 9), functions (8.3.5), and objects (Clause 7).

3 Annex B recommends lower bounds on the capacity of conforming implementations.

4 Annex C summarizes the evolution of C++ since its first published description, and explains in detail the differences between C++ and C. Certain features of C++ exist solely for compatibility purposes; Annex D describes those features.

5 Throughout this International Standard, each example is introduced by "[*Example:*" and terminated by "— *end example*]". Each note is introduced by "[*Note:*" and terminated by "— *end note*]". Examples and notes may be nested.

1.6 Syntax notation [syntax]

1 In the syntax notation used in this International Standard, syntactic categories are indicated by *italic* type, and literal words and characters in `constant width` type. Alternatives are listed on separate lines except in a few cases where a long set of alternatives is presented on one line, marked by the phrase "one of." An optional terminal or nonterminal symbol is indicated by the subscript "$_{opt}$ ", so

`{` *expression*$_{opt}$ `}`

indicates an optional expression enclosed in braces.

2 Names for syntactic categories have generally been chosen according to the following rules:

— *X-name* is a use of an identifier in a context that determines its meaning (e.g. *class-name*, *typedef-name*).

— *X-id* is an identifier with no context-dependent meaning (e.g. *qualified-id*).

— *X-seq* is one or more *X*'s without intervening delimiters (e.g. *declaration-seq* is a sequence of declarations).

— *X-list* is one or more *X*'s separated by intervening commas (e.g. *expression-list* is a sequence of expressions separated by commas).

3) This documentation also defines implementation-defined behavior; see 1.9.

1.7 The C++ memory model [intro.memory]

1 The fundamental storage unit in the C++ memory model is the *byte.* A byte is at least large enough to contain any member of the basic execution character set and the eight-bit code units of the Unicode UTF-8 encoding form and is composed of a contiguous sequence of bits, the number of which is implementation-defined. The least significant bit is called the *low-order* bit; the most significant bit is called the *high-order* bit. The memory available to a C++ program consists of one or more sequences of contiguous bytes. Every byte has a unique address.

2 [*Note:* the representation of types is described in 3.9. — *end note*]

3 A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having non-zero width. [*Note:* Various features of the language, such as references and virtual functions, might involve additional memory locations that are not accessible to programs but are managed by the implementation. — *end note*] Two threads of execution can update and access separate memory locations without interfering with each other.

4 [*Note:* Thus a bit-field and an adjacent non-bit-field are in separate memory locations, and therefore can be concurrently updated by two threads of execution without interference. The same applies to two bit-fields, if one is declared inside a nested struct declaration and the other is not, or if the two are separated by a zero-length bit-field declaration, or if they are separated by a non-bit-field declaration. It is not safe to concurrently update two bit-fields in the same struct if all fields between them are also bit-fields, no matter what the sizes of those intervening bit-fields happen to be. — *end note*]

5 [*Example:* A structure declared as

```
struct {
  char a;
  int b:5,
  c:11,
  :0,
  d:8;
  struct {int ee:8;} e;
}
```

contains four separate memory locations: The field `a` and bit-fields `d` and `e.ee` are each separate memory locations, and can be modified concurrently without interfering with each other. The bit-fields `b` and `c` together constitute the fourth memory location. The bit-fields `b` and `c` cannot be concurrently modified, but `b` and `a`, for example, can be. — *end example*]

1.8 The C++ object model [intro.object]

1 The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An *object* is a region of storage. [*Note:* A function is not an object, regardless of whether or not it occupies storage in the way that objects do. — *end note*] An object is created by a *definition* (3.1), by a *new-expression* (5.3.4) or by the implementation (12.2) when needed. The properties of an object are determined when the object is created. An object can have a *name* (Clause 3). An object has a *storage duration* (3.7) which influences its *lifetime* (3.8). An object has a *type* (3.9). The term *object type* refers to the type with which the object is created. Some objects are *polymorphic* (10.3); the implementation generates information associated with each such object that makes it possible to determine that object's type during program execution. For other objects, the interpretation of the values found therein is determined by the type of the *expression*s (Clause 5) used to access them.

2 Objects can contain other objects, called *subobjects*. A subobject can be a *member subobject* (9.2), a *base class subobject* (Clause 10), or an array element. An object that is not a subobject of any other object is called a *complete object*.

3 For every object `x`, there is some object called *the complete object of* `x`, determined as follows:

— If `x` is a complete object, then `x` is the complete object of `x`.

— Otherwise, the complete object of `x` is the complete object of the (unique) object that contains `x`.

4 If a complete object, a data member (9.2), or an array element is of class type, its type is considered the *most derived* class, to distinguish it from the class type of any base class subobject; an object of a most derived class type or of a non-class type is called a *most derived object*.

5 Unless it is a bit-field (9.6), a most derived object shall have a non-zero size and shall occupy one or more bytes of storage. Base class subobjects may have zero size. An object of trivially copyable or standard-layout type (3.9) shall occupy contiguous bytes of storage.

6 [*Note:* C++ provides a variety of built-in types and several ways of composing new types from existing types (3.9). — *end note*]

1.9 Program execution [intro.execution]

1 The semantic descriptions in this International Standard define a parameterized nondeterministic abstract machine. This International Standard places no requirement on the structure of conforming implementations. In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.[4]

2 Certain aspects and operations of the abstract machine are described in this International Standard as implementation-defined (for example, `sizeof(int)`). These constitute the parameters of the abstract machine. Each implementation shall include documentation describing its characteristics and behavior in these respects.[5] Such documentation shall define the instance of the abstract machine that corresponds to that implementation (referred to as the "corresponding instance" below).

3 Certain other aspects and operations of the abstract machine are described in this International Standard as unspecified (for example, order of evaluation of arguments to a function). Where possible, this International Standard defines a set of allowable behaviors. These define the nondeterministic aspects of the abstract machine. An instance of the abstract machine can thus have more than one possible execution sequence for a given program and a given input.

4 Certain other operations are described in this International Standard as undefined (for example, the effect of dereferencing the null pointer). [*Note:* this International Standard imposes no requirements on the behavior of programs that contain undefined behavior. — *end note*]

5 A conforming implementation executing a well-formed program shall produce the same observable behavior as one of the possible execution sequences of the corresponding instance of the abstract machine with the same program and the same input. However, if any such execution sequence contains an undefined operation, this International Standard places no requirement on the implementation executing that program with that input (not even with regard to operations preceding the first undefined operation).

4) This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is *as if* the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.

5) This documentation also includes conditonally-supported constructs and locale-specific behavior. See 1.4.

6 The observable behavior of the abstract machine is its sequence of reads and writes to `volatile` data and calls to library I/O functions.[6]

7 When the processing of the abstract machine is interrupted by receipt of a signal, the values of objects which are neither

— of type `volatile std::sig_atomic_t` nor

— lock-free atomic objects (29.2)

are unspecified, and the value of any object not in either of these two categories that is modified by the handler becomes undefined.

8 An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).

9 The least requirements on a conforming implementation are:

— Access to volatile objects are evaluated strictly according to the rules of the abstract machine.

— At program termination, all data written into files shall be identical to one of the possible results that execution of the program according to the abstract semantics would have produced.

— The input and output dynamics of interactive devices shall take place in such a fashion that prompting messages actually appear prior to a program waiting for input. What constitutes an interactive device is implementation-defined.

[*Note:* more stringent correspondences between abstract and actual semantics may be defined by each implementation. — *end note*]

10 [*Note:* operators can be regrouped according to the usual mathematical rules only where the operators really are associative or commutative.[7] For example, in the following fragment

```
int a, b;
/* ... */
a = a + 32760 + b + 5;
```

the expression statement behaves exactly the same as

```
a = (((a + 32760) + b) + 5);
```

due to the associativity and precedence of these operators. Thus, the result of the sum `(a + 32760)` is next added to `b`, and that result is then added to 5 which results in the value assigned to `a`. On a machine in which overflows produce an exception and in which the range of values representable by an `int` is `[-32768,+32767]`, the implementation cannot rewrite this expression as

```
a = ((a + b) + 32765);
```

since if the values for `a` and `b` were, respectively, -32754 and -15, the sum `a + b` would produce an exception while the original expression would not; nor can the expression be rewritten either as

```
a = ((a + 32765) + b);
```

or

6) An implementation can offer additional library I/O functions as an extension. Implementations that do so should treat calls to those functions as "observable behavior" as well.

7) Overloaded operators are never assumed to be associative or commutative.

```
a = (a + (b + 32765));
```

since the values for `a` and `b` might have been, respectively, 4 and -8 or -17 and 12. However on a machine in which overflows do not produce an exception and in which the results of overflows are reversible, the above expression statement can be rewritten by the implementation in any of the above ways because the same result will occur. — *end note*]

11 A *full-expression* is an expression that is not a subexpression of another expression. If a language construct is defined to produce an implicit call of a function, a use of the language construct is considered to be an expression for the purposes of this definition. A call to a destructor generated at the end of the lifetime of an object other than a temporary object is an implicit full-expression. Conversions applied to the result of an expression in order to satisfy the requirements of the language construct in which the expression appears are also considered to be part of the full-expression.

[*Example:*

```
struct S {
  S(int i): I(i) { }
  int& v() { return I; }
private:
  int I;
};

 S s1(1);            // full-expression is call of S::S(int)
 S s2 = 2;           // full-expression is call of S::S(int)

void f() {
  if (S(3).v())      // full-expression includes lvalue-to-rvalue and
                     // int to bool conversions, performed before
                     // temporary is deleted at end of full-expression
  { }
}
```

— *end example*]

12 [*Note:* the evaluation of a full-expression can include the evaluation of subexpressions that are not lexically part of the full-expression. For example, subexpressions involved in evaluating default argument expressions (8.3.6) are considered to be created in the expression that calls the function, not the expression that defines the default argument. — *end note*]

13 Accessing an object designated by a `volatile` lvalue (3.10), modifying an object, calling a library I/O function, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. *Evaluation* of an expression (or a sub-expression) in general includes both value computations (including determining the identity of an object for lvalue evaluation and fetching a value previously assigned to an object for rvalue evaluation) and initiation of side effects. When a call to a library I/O function returns or an access to a `volatile` object is evaluated the side effect is considered complete, even though some external actions implied by the call (such as the I/O itself) or by the `volatile` access may not have completed yet.

14 *Sequenced before* is an asymmetric, transitive, pair-wise relation between evaluations executed by a single thread, which induces a partial order among those evaluations. Given any two evaluations A and B, if A is sequenced before B, then the execution of A shall precede the execution of B. If A is not sequenced before B and B is not sequenced before A, then A and B are *unsequenced*. [*Note:* The execution of unsequenced evaluations can overlap. — *end note*] Evaluations A and B are *indeterminately sequenced* when either A

is sequenced before B or B is sequenced before A, but it is unspecified which. [*Note:* Indeterminately sequenced evaluations cannot overlap, but either could be executed first. — *end note*]

15 Every value computation and side effect associated with a full-expression is sequenced before every value computation and side effect associated with the next full-expression to be evaluated.[8].

16 Except where noted, evaluations of operands of individual operators and of subexpressions of individual expressions are unsequenced. [*Note:* In an expression that is evaluated more than once during the execution of a program, unsequenced and indeterminately sequenced evaluations of its subexpressions need not be performed consistently in different evaluations. — *end note*] The value computations of the operands of an operator are sequenced before the value computation of the result of the operator. If a side effect on a scalar object is unsequenced relative to either another side effect on the same scalar object or a value computation using the value of the same scalar object, the behavior is undefined.

[*Example:*

```
void f(int, int);
void g(int i, int *v) {
  i = v[i++];         // the behavior is undefined
  i = 7, i++, i++;    // i becomes 9

  i = i++ + 1;        // the behavior is undefined
  i = i + 1;          // the value of i is incremented

  f(i = -1, i = -1);  // the behavior is undefined
}
```

— *end example*]

When calling a function (whether or not the function is inline), every value computation and side effect associated with any argument expression, or with the postfix expression designating the called function, is sequenced before execution of every expression or statement in the body of the called function. [*Note:* Value computations and side effects associated with different argument expressions are unsequenced. — *end note*] Every evaluation in the calling function (including other function calls) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function.[9] Several contexts in C++ cause evaluation of a function call, even though no corresponding function call syntax appears in the translation unit. [*Example:* Evaluation of a `new` expression invokes one or more allocation and constructor functions; see 5.3.4. For another example, invocation of a conversion function (12.3.2) can arise in contexts in which no function call syntax appears. — *end example*] The sequencing constraints on the execution of the called function (as described above) are features of the function calls as evaluated, whatever the syntax of the expression that calls the function might be.

1.10 Multi-threaded executions and data races [intro.multithread]

1 Under a hosted implementation, a C++ program can have more than one *thread of execution* (a.k.a. *thread*) running concurrently. The execution of each thread proceeds as defined by the remainder of this standard. The execution of the entire program consists of an execution of all of its threads. [*Note:* Usually the execution can be viewed as an interleaving of all its threads. However, some kinds of atomic operations, for example, allow executions inconsistent with a simple interleaving, as described below. — *end note*] Under

8) As specified in 12.2, after a full-expression is evaluated, a sequence of zero or more invocations of destructor functions for temporary objects takes place, usually in reverse order of the construction of each temporary object.

9) In other words, function executions do not interleave with each other.

a freestanding implementation, it is implementation-defined whether a program can have more than one thread of execution.

2 The value of an object visible to a thread T at a particular point might be the initial value of the object, a value assigned to the object by T, or a value assigned to the object by another thread, according to the rules below. [*Note:* In some cases, there may instead be undefined behavior. Much of this section is motivated by the desire to support atomic operations with explicit and detailed visibility constraints. However, it also implicitly supports a simpler view for more restricted programs. — *end note*]

3 Two expression evaluations *conflict* if one of them modifies a memory location and the other one accesses or modifies the same memory location.

4 The library defines a number of atomic operations (Clause 29) and operations on locks (Clause 30) that are specially identified as synchronization operations. These operations play a special role in making assignments in one thread visible to another. A synchronization operation on one or more memory locations is either a consume operation, an acquire operation, a release operation, or both an acquire and release operation. A synchronization operation without an associated memory location is a fence and can be either an acquire fence, a release fence, or both an acquire and release fence. In addition, there are relaxed atomic operations, which are not synchronization operations, and atomic read-modify-write operations, which have special characteristics. [*Note:* For example, a call that acquires a lock will perform an acquire operation on the locations comprising the lock. Correspondingly, a call that releases the same lock will perform a release operation on those same locations. Informally, performing a release operation on A forces prior side effects on other memory locations to become visible to other threads that later perform a consume or an acquire operation on A. We do not include "relaxed" atomic operations as synchronization operations although, like synchronization operations, they cannot contribute to data races. — *end note*]

5 All modifications to a particular atomic object M occur in some particular total order, called the *modification order* of M. If A and B are modifications of an atomic object M and A happens before (as defined below) B, then A shall precede B in the modification order of M, which is defined below. [*Note:* This states that the modification orders must respect *happens before*. — *end note*] [*Note:* There is a separate order for each scalar object. There is no requirement that these can be combined into a single total order for all objects. In general this will be impossible since different threads may observe modifications to different variables in inconsistent orders. — *end note*]

6 A *release sequence* on an atomic object M is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first operation is a release, and every subsequent operation

- is performed by the same thread that performed the release, or
- is an atomic read-modify-write operation.

7 Certain library calls *synchronize with* other library calls performed by another thread. In particular, an atomic operation A that performs a release operation on an object M synchronizes with an atomic operation B that performs an acquire operation on M and reads a value written by any side effect in the release sequence headed by A. [*Note:* Except in the specified cases, reading a later value does not necessarily ensure visibility as described below. Such a requirement would sometimes interfere with efficient implementation. — *end note*] [*Note:* The specifications of the synchronization operations define when one reads the value written by another. For atomic variables, the definition is clear. All operations on a given lock occur in a single total order. Each lock acquisition "reads the value written" by the last lock release. — *end note*]

8 An evaluation A *carries a dependency* to an evaluation B if

- the value of A is used as an operand of B, unless:
  - B is an invocation of any specialization of `std::kill_dependency` (29.1), or

— A is the left operand of a built-in logical AND (`&&`, see 5.14) or logical OR (`||`, see 5.15) operator, or

— A is the left operand of a conditional (`?:`, see 5.16) operator, or

— A is the left operand of the built-in comma (`,`) operator (5.18);

or

— A writes a scalar object or bit-field M, B reads the value written by A from M, and A is sequenced before B, or

— for some evaluation X, A carries a dependency to X, and X carries a dependency to B.

[*Note:* "Carries a dependency to" is a subset of "is sequenced before", and is similarly strictly intra-thread. — *end note*]

9 An evaluation A is *dependency-ordered before* an evaluation B if

— A performs a release operation on an atomic object M, and B performs a consume operation on M and reads a value written by any side effect in the release sequence headed by A, or

— for some evaluation X, A is dependency-ordered before X and X carries a dependency to B.

[*Note:* The relation "is dependency-ordered before" is analogous to "synchronizes with", but uses release/-consume in place of release/acquire. — *end note*]

10 An evaluation A *inter-thread happens before* an evaluation B if

— A synchronizes with B, or

— A is dependency-ordered before B, or

— for some evaluation X

— A synchronizes with X and X is sequenced before B, or

— A is sequenced before X and X inter-thread happens before B, or

— A inter-thread happens before X and X inter-thread happens before B.

[*Note:* The "inter-thread happens before" relation describes arbitrary concatenations of "sequenced before", "synchronizes with" and "dependency-ordered before" relationships, with two exceptions. The first exception is that a concatenation is not permitted to end with "dependency-ordered before" followed by "sequenced before". The reason for this limitation is that a consume operation participating in a "dependency-ordered before" relationship provides ordering only with respect to operations to which this consume operation actually carries a dependency. The reason that this limitation applies only to the end of such a concatenation is that any subsequent release operation will provide the required ordering for a prior consume operation. The second exception is that a concatenation is not permitted to consist entirely of "sequenced before". The reasons for this limitation are (1) to permit "inter-thread happens before" to be transitively closed and (2) the "happens before" relation, defined below, provides for relationships consisting entirely of "sequenced before". — *end note*]

11 An evaluation A *happens before* an evaluation B if:

— A is sequenced before B, or

— A inter-thread happens before B.

12 A *visible side effect* A on an object M with respect to a value computation B of M satisfies the conditions:

— A happens before B, and

— there is no other side effect X to M such that A happens before X and X happens before B.

The value of a non-atomic scalar object M, as determined by evaluation B, shall be the value stored by the visible side effect A. [*Note:* If there is ambiguity about which side effect to a non-atomic object is visible, then there is a data race, and the behavior is undefined. — *end note*] [*Note:* This states that operations on ordinary variables are not visibly reordered. This is not actually detectable without data races, but it is necessary to ensure that data races, as defined here, and with suitable restrictions on the use of atomics, correspond to data races in a simple interleaved (sequentially consistent) execution. — *end note*]

13 The *visible sequence of side effects* on an atomic object M, with respect to a value computation B of M, is a maximal contiguous sub-sequence of side effects in the modification order of M, where the first side effect is visible with respect to B, and for every subsequent side effect, it is not the case that B happens before it. The value of an atomic object M, as determined by evaluation B, shall be the value stored by some operation in the visible sequence of M with respect to B. Furthermore, if a value computation A of an atomic object M happens before a value computation B of M, and the value computed by A corresponds to the value stored by side effect X, then the value computed by B shall either equal the value computed by A, or be the value stored by side effect Y, where Y follows X in the modification order of M. [*Note:* This effectively disallows compiler reordering of atomic operations to a single object, even if both operations are "relaxed" loads. By doing so, we effectively make the "cache coherence" guarantee provided by most hardware available to C++ atomic operations. — *end note*] [*Note:* The visible sequence depends on the *happens before* relation, which depends on the values observed by loads of atomics, which we are restricting here. The intended reading is that there must exist an association of atomic loads with modifications they observe that, together with suitably chosen modification orders and the *happens before* relation derived as described above, satisfy the resulting constraints as imposed here. — *end note*]

14 The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. [*Note:* It can be shown that programs that correctly use simple locks to prevent all data races, and use no other synchronization operations, behave as though the executions of their constituent threads were simply interleaved, with each observed value of an object being the last value assigned in that interleaving. This is normally referred to as "sequential consistency". However, this applies only to race-free programs, and race-free programs cannot observe most program transformations that do not change single-threaded program semantics. In fact, most single-threaded program transformations continue to be allowed, since any program that behaves differently as a result must perform an undefined operation. — *end note*]

15 [*Note:* Compiler transformations that introduce assignments to a potentially shared memory location that would not be modified by the abstract machine are generally precluded by this standard, since such an assignment might overwrite another assignment by a different thread in cases in which an abstract machine execution would not have encountered a data race. This includes implementations of data member assignment that overwrite adjacent members in separate memory locations. We also generally preclude reordering of atomic loads in cases in which the atomics in question may alias, since this may violate the "visible sequence" rules. — *end note*]

16 [*Note:* Transformations that introduce a speculative read of a potentially shared memory location may not preserve the semantics of the C++ program as defined in this standard, since they potentially introduce a data race. However, they are typically valid in the context of an optimizing compiler that targets a specific machine with well-defined semantics for data races. They would be invalid for a hypothetical machine that is not tolerant of races or provides hardware race detection. — *end note*]

1.11 Acknowledgments [intro.ack]

1 The C++ programming language as described in this International Standard is based on the language as described in Chapter R (Reference Manual) of Stroustrup: *The C++ Programming Language* (second edition,

Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T). That, in turn, is based on the C programming language as described in Appendix A of Kernighan and Ritchie: *The C Programming Language* (Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T).

2 Portions of the library Clauses of this International Standard are based on work by P.J. Plauger, which was published as *The Draft Standard C++ Library* (Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger).

2 Lexical conventions [lex]

1 The text of the program is kept in units called *source files* in this International Standard. A source file together with all the headers (17.6.2.3) and source files included (16.2) via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion (16.1) preprocessing directives, is called a *translation unit*. [*Note:* a C++ program need not all be translated at the same time. — *end note*]

2 [*Note:* previously translated translation units and instantiation units can be preserved individually or in libraries. The separate translation units of a program communicate (3.5) by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. Translation units can be separately translated and then later linked to produce an executable program (3.5). — *end note*]

2.1 Phases of translation [lex.phases]

1 The precedence among the syntax rules of translation is specified by the following phases.[10]

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. The set of physical source file characters accepted is implementation-defined. Trigraph sequences (2.3) are replaced by corresponding single-character internal representations. Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character. (An implementation may use any internal encoding, so long as an actual extended character encountered in the source file, and the same extended character expressed in the source file as a universal-character-name (i.e. using the `\uXXXX` notation), are handled equivalently.)

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. Only the last backslash on any physical source line shall be eligible for being part of such a splice. If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined. If a source file that is not empty does not end in a new-line character, or ends in a new-line character immediately preceded by a backslash character before any such splicing takes place, the behavior is undefined.

3. The source file is decomposed into preprocessing tokens (2.4) and sequences of white-space characters (including comments). A source file shall not end in a partial preprocessing token or in a partial comment.[11] Each comment is replaced by one space character. New-line characters are retained. Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined. The process of dividing a source file's characters into preprocessing tokens is context-dependent. [*Example:* see the handling of `<` within a `#include` preprocessing directive. — *end example*]

4. Preprocessing directives are executed, macro invocations are expanded, and `_Pragma` unary operator expressions are executed. If a character sequence that matches the syntax of a universal-character-name is produced by token concatenation (16.3.3), the behavior is undefined. A `#include` preprocessing di-

10) Implementations must behave as if these separate phases occur, although in practice different phases might be folded together.

11) A partial preprocessing token would arise from a source file ending in the first portion of a multi-character token that requires a terminating sequence of characters, such as a *header-name* that is missing the closing `"` or `>`. A partial comment would arise from a source file ending with an unclosed `/*` comment.

rective causes the named header or source file to be processed from phase 1 through phase 4, recursively. All preprocessing directives are then deleted.

5. Each source character set member and universal-character-name in a character literal or a string literal, as well as each escape sequence in a character literal or a non-raw string literal, is converted to the corresponding member of the execution character set (2.13.2, 2.13.4); if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[12]

6. Adjacent literal tokens are concatenated.

7. White-space characters separating tokens are no longer significant. Each preprocessing token is converted into a token. (2.6). The resulting tokens are syntactically and semantically analyzed and translated as a translation unit. [*Note:* The process of analyzing and translating the tokens may occasionally result in one token being replaced by a sequence of other tokens (14.2). — *end note*] [*Note:* Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation. The description is conceptual only, and does not specify any particular implementation. — *end note*]

8. Translated translation units and instantiation units are combined as follows: [*Note:* some or all of these may be supplied from a library. — *end note*] Each translated translation unit is examined to produce a list of required instantiations. [*Note:* this may include instantiations which have been explicitly requested (14.7.2). — *end note*] The definitions of the required templates are located. It is implementation-defined whether the source of the translation units containing these definitions is required to be available. [*Note:* an implementation could encode sufficient information into the translated translation unit so as to ensure the source is not required here. — *end note*] All the required instantiations are performed to produce *instantiation units*. [*Note:* these are similar to translated translation units, but contain no references to uninstantiated templates and no template definitions. — *end note*] The program is ill-formed if any instantiation fails.

9. All external object and function references are resolved. Library components are linked to satisfy external references to functions and objects not defined in the current translation. All such translator output is collected into a program image which contains information needed for execution in its execution environment.

2.2 Character sets [lex.charset]

1 The *basic source character set* consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line, plus the following 91 graphical characters:[13]

```
a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

0 1 2 3 4 5 6 7 8 9

_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

2 The *universal-character-name* construct provides a way to name other characters.

hex-quad:
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

12) An implementation need not convert all non-corresponding source characters to the same execution character.

13) The glyphs for the members of the basic source character set are intended to identify characters from the subset of ISO/IEC 10646 which corresponds to the ASCII character set. However, because the mapping from source file characters to the source character set (described in translation phase 1) is specified as implementation-defined, an implementation is required to document how the basic source characters are represented in source files.

universal-character-name:
\u *hex-quad*
\U *hex-quad hex-quad*

The character designated by the universal-character-name \UNNNNNNNN is that character whose character short name in ISO/IEC 10646 is NNNNNNNN; the character designated by the universal-character-name \uNNNN is that character whose character short name in ISO/IEC 10646 is 0000NNNN. If the hexadecimal value for a universal-character-name corresponds to a surrogate code point (in the range 0xD800–0xDFFF, inclusive), the program is ill-formed. Additionally, if the hexadecimal value for a universal-character-name outside a character or string literal corresponds to a control character (in either of the ranges 0x00–0x1F or 0x7F–0x9F, both inclusive) or to a character in the basic source character set, the program is ill-formed.

3 The *basic execution character set* and the *basic execution wide-character set* shall each contain all the members of the basic source character set, plus control characters representing alert, backspace, and carriage return, plus a *null character* (respectively, *null wide character*), whose representation has all zero bits. For each basic execution character set, the values of the members shall be non-negative and distinct from one another. In both the source and execution basic character sets, the value of each character after 0 in the above list of decimal digits shall be one greater than the value of the previous. The *execution character set* and the *execution wide-character set* are supersets of the basic execution character set and the basic execution wide-character set, respectively. The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

2.3 Trigraph sequences [lex.trigraph]

1 Before any other processing takes place, each occurrence of one of the following sequences of three characters ("*trigraph sequences*") is replaced by the single character indicated in Table 1.

Table 1 — Trigraph sequences

Trigraph	Replacement	Trigraph	Replacement	Trigraph	Replacement
??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	~

2 [*Example:*

```
??=define arraycheck(a,b) a??(b??) ??!??! b??(a??)
```

becomes

```
#define arraycheck(a,b) a[b] || b[a]
```

— *end example*]

3 No other trigraph sequence exists. Each ? that does not begin one of the trigraphs listed above is not changed.

2.4 Preprocessing tokens [lex.pptoken]

preprocessing-token:
header-name
identifier
pp-number
character-literal
user-defined-character-literal
string-literal
user-defined-string-literal
preprocessing-op-or-punc
each non-white-space character that cannot be one of the above

1 Each preprocessing token that is converted to a token (2.6) shall have the lexical form of a keyword, an identifier, a literal, an operator, or a punctuator.

2

A preprocessing token is the minimal lexical element of the language in translation phases 3 through 6. The categories of preprocessing token are: header names, identifiers, preprocessing numbers, character literals (including user-defined character literals), string literals (including user-defined string literals), preprocessing operators and punctuators, and single non-white-space characters that do not lexically match the other preprocessing token categories. If a `'` or a `"` character matches the last category, the behavior is undefined. Preprocessing tokens can be separated by white space; this consists of comments (2.7), or white-space characters (space, horizontal tab, new-line, vertical tab, and form-feed), or both. As described in Clause 16, in certain circumstances during translation phase 4, white space (or the absence thereof) serves as more than preprocessing token separation. White space can appear within a preprocessing token only as part of a header name or between the quotation characters in a character literal or string literal.

3 If the input stream has been parsed into preprocessing tokens up to a given character, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail.

4 [*Example:* The program fragment `1Ex` is parsed as a preprocessing number token (one that is not a valid floating or integer literal token), even though a parse as the pair of preprocessing tokens `1` and `Ex` might produce a valid expression (for example, if `Ex` were a macro defined as `+1`). Similarly, the program fragment `1E1` is parsed as a preprocessing number (one that is a valid floating literal token), whether or not `E` is a macro name. — *end example*]

5 [*Example:* The program fragment `x+++++y` is parsed as `x ++ ++ + y`, which, if `x` and `y` are of built-in types, violates a constraint on increment operators, even though the parse `x ++ + ++ y` might yield a correct expression. — *end example*]

2.5 Alternative tokens [lex.digraph]

1 Alternative token representations are provided for some operators and punctuators.[14]

2 In all respects of the language, each alternative token behaves the same, respectively, as its primary token,

Table 2 — Alternative tokens

Alternative	Primary	Alternative	Primary	Alternative	Primary
<%	{	and	&&	and_eq	&=
%>	}	bitor	|	or_eq	|=
<:	[	or	||	xor_eq	^=
:>	]	xor	^	not	!
%:	#	compl	~	not_eq	!=
%:%:	##	bitand	&		

except for its spelling.[15] The set of alternative tokens is defined in Table 2.

2.6 Tokens [lex.token]

token:
identifier
keyword
literal
operator
punctuator

1 There are five kinds of tokens: identifiers, keywords, literals,[16] operators, and other separators. Blanks, horizontal and vertical tabs, newlines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens. [*Note:* Some white space is required to separate otherwise adjacent identifiers, keywords, numeric literals, and alternative tokens containing alphabetic characters. — *end note*]

2.7 Comments [lex.comment]

1 The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates with the next new-line character. If there is a form-feed or a vertical-tab character in such a comment, only white-space characters shall appear between it and the new-line that terminates the comment; no diagnostic is required. [*Note:* The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment. — *end note*]

2.8 Header names [lex.header]

header-name:
`<` *h-char-sequence* `>`
`"` *q-char-sequence* `"`

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any member of the source character set except new-line and >

14) These include "digraphs" and additional reserved words. The term "digraph" (token consisting of two characters) is not perfectly descriptive, since one of the alternative preprocessing-tokens is `%:%:` and of course several primary tokens contain two characters. Nonetheless, those alternative tokens that aren't lexical keywords are colloquially known as "digraphs".

15) Thus the "stringized" values (16.3.2) of `[` and `<:` will be different, maintaining the source spelling, but the tokens can otherwise be freely interchanged.

16) Literals include strings and character and numeric literals.

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any member of the source character set except new-line and "

1 Header name preprocessing tokens shall only appear within a `#include` preprocessing directive (16.2). The sequences in both forms of *header-name*s are mapped in an implementation-defined manner to headers or to external source file names as specified in 16.2.

2 If either of the characters `'` or `\`, or either of the character sequences `/*` or `//` appears in a *q-char-sequence* or a *h-char-sequence*, or the character `"` appears in a *h-char-sequence*, the behavior is undefined.[17]

2.9 Preprocessing numbers [lex.ppnumber]

pp-number:
digit
`.` *digit*
pp-number digit
pp-number identifier-nondigit
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `.`

1 Preprocessing number tokens lexically include all integral literal tokens (2.13.1) and all floating literal tokens (2.13.3).

2 A preprocessing number does not have a type or a value; it acquires both after a successful conversion (as part of translation phase 7, 2.1) to an integral literal token or a floating literal token.

2.10 Identifiers [lex.name]

identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters

nondigit: one of
`a b c d e f g h i j k l m`
`n o p q r s t u v w x y z`
`A B C D E F G H I J K L M`
`N O P Q R S T U V W X Y Z _`

digit: one of
`0 1 2 3 4 5 6 7 8 9`

1 An identifier is an arbitrarily long sequence of letters and digits. Each universal-character-name in an identifier shall designate a character whose encoding in ISO 10646 falls into one of the ranges specified in Annex A of TR 10176:2003. Upper- and lower-case letters are different. All characters are significant.[18]

17) Thus, sequences of characters that resemble escape sequences cause undefined behavior.

18) On systems in which linkers cannot accept extended characters, an encoding of the universal-character-name may be used in forming valid external identifiers. For example, some otherwise unused character or sequence of characters may be used to encode the `\u` in a universal-character-name. Extended characters may produce a long external identifier, but C++ does not place a translation limit on significant characters for external identifiers. In C++, upper- and lower-case letters are considered different for all identifiers, including external identifiers.

2 In addition, some identifiers are reserved for use by C++ implementations and standard libraries (17.6.4.3.3) and shall not be used otherwise; no diagnostic is required.

2.11 Keywords [lex.key]

1 The identifiers shown in Table 3 are reserved for use as keywords (that is, they are unconditionally treated as keywords in phase 7) except in an *attribute-token* (7.6.1):

Table 3 — Keywords

const	for	register	true	
alignof	const_cast	friend	reinterpret_cast	try
asm	continue	goto	requires	typedef
auto	decltype	if	return	typeid
axiom	default	inline	short	typename
bool	delete	int	signed	union
break	double	late_check	sizeof	unsigned
case	do	long	static	using
catch	dynamic_cast	mutable	static_assert	virtual
char16_t	else	namespace	static_cast	void
char32_t	enum	new	struct	volatile
char	explicit	nullptr	switch	wchar_t
class	export	operator	template	while
concept	extern	private	this	
concept_map	false	protected	thread_local	
constexpr	float	public	throw	

2 Furthermore, the alternative representations shown in Table 4 for certain operators and punctuators (2.5) are reserved and shall not be used otherwise:

Table 4 — Alternative representations

and	and_eq	bitand	bitor	compl	not
not_eq	or	or_eq	xor	xor_eq	

2.12 Operators and punctuators [lex.operators]

1 The lexical representation of C++ programs includes a number of preprocessing tokens which are used in the syntax of the preprocessor or are converted into tokens for operators and punctuators:

preprocessing-op-or-punc: one of

{	}	[	]	#	##	(	)	
<:	:>	<%	%>	%:	%:%:	;	:	...
new	delete	?	::	.	.*			
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->
and	and_eq	bitand	bitor	compl	not	not_eq		
or	or_eq	xor	xor_eq					

Each *preprocessing-op-or-punc* is converted to a single token in translation phase 7 (2.1).

2.13 Literals [lex.literal]

1 There are several kinds of literals.[19]

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal
pointer-literal
user-defined-literal

2.13.1 Integer literals [lex.icon]

integer-literal:
decimal-literal integer-suffix$_{opt}$
octal-literal integer-suffix$_{opt}$
hexadecimal-literal integer-suffix$_{opt}$

decimal-literal:
nonzero-digit
decimal-literal digit

octal-literal:
`0`
octal-literal octal-digit

hexadecimal-literal:
`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
unsigned-suffix long-long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l L`

long-long-suffix: one of
`ll LL`

1 An *integer literal* is a sequence of digits that has no period or exponent part. An integer literal may have a prefix that specifies its base and a suffix that specifies its type. The lexically first digit of the sequence

19) The term "literal" generally designates, in this International Standard, those tokens that are called "constants" in ISO C.

of digits is the most significant. A *decimal* integer literal (base ten) begins with a digit other than `0` and consists of a sequence of decimal digits. An *octal* integer literal (base eight) begins with the digit `0` and consists of a sequence of octal digits.[20] A *hexadecimal* integer literal (base sixteen) begins with `0x` or `0X` and consists of a sequence of hexadecimal digits, which include the decimal digits and the letters `a` through `f` and `A` through `F` with decimal values ten through fifteen. [*Example:* the number twelve can be written `12`, `014`, or `0XC`. — *end example*]

2 The type of an integer literal is the first of the corresponding list in Table 5 in which its value can be represented.

Table 5 — Types of integer constants

Suffix	Decimal constant	Octal or hexadecimal constant

none	`int`
	`long int`
	`long long int`
		`int`
		`unsigned int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long long int`

`u` or `U`	`unsigned int`
	`unsigned long int`
	`unsigned long long int`
		`unsigned int`
		`unsigned long int`
		`unsigned long long int`

`l` or `L`	`long int`
	`long long int`
		`long int`
		`unsigned long int`
		`long long int`
		`unsigned long int`

Both `u` or `U` and `l` or `L`	`unsigned long int` `unsigned long long int`	`unsigned long int` `unsigned long long int`
`ll` or `LL`	`long long int`	`long long int` `unsigned long int`
Both `u` or `U` and `ll` or `LL`	`unsigned long long int`	`unsigned long long int`

3 If an integer literal cannot be represented by any type in its list and an extended integer type can represent its value, it may have that extended integer type. If all of the types in the list for the literal are signed, the extended integer type shall be signed. If all of the types in the list for the literal are unsigned, the extended integer type shall be unsigned. If the list contains both signed and unsigned types, the extended integer type may be signed or unsigned. A program is ill-formed if one of its translation units contains an integer literal that cannot be represented by any of the allowed types.

2.13.2 Character literals [lex.ccon]

character-literal:
' *c-char-sequence* '
u' *c-char-sequence* '
U' *c-char-sequence* '
L' *c-char-sequence* '

c-char-sequence:
c-char
c-char-sequence c-char

20) The digits `8` and `9` are not octal digits.

c-char:
any member of the source character set except
the single-quote ', backslash \, or new-line character
escape-sequence
universal-character-name

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
`\'` `\"` `\?` `\\`
`\a` `\b` `\f` `\n` `\r` `\t` `\v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

1 A character literal is one or more characters enclosed in single quotes, as in `'x'`, optionally preceded by one of the letters `u`, `U`, or `L`, as in `u'y'`, `U'z'`, or `L'x'`, respectively. A character literal that does not begin with `u`, `U`, or `L` is an ordinary character literal, also referred to as a narrow-character literal. An ordinary character literal that contains a single *c-char* has type `char`, with value equal to the numerical value of the encoding of the *c-char* in the execution character set. An ordinary character literal that contains more than one *c-char* is a *multicharacter literal*. A multicharacter literal has type `int` and implementation-defined value.

2 A character literal that begins with the letter `u`, such as `u'y'`, is a character literal of type `char16_t`. The value of a `char16_t` literal containing a single *c-char* is equal to its ISO 10646 code point value, provided that the code point is representable with a single 16-bit code unit. (That is, provided it is a basic multi-lingual plane code point.) If the value is not representable within 16 bits, the program is ill-formed. A `char16_t` literal containing multiple *c-char*s is ill-formed. A character literal that begins with the letter `U`, such as `U'z'`, is a character literal of type `char32_t`. The value of a `char32_t` literal containing a single *c-char* is equal to its ISO 10646 code point value. A `char32_t` literal containing multiple *c-char*s is ill-formed. A character literal that begins with the letter `L`, such as `L'x'`, is a wide-character literal. A wide-character literal has type `wchar_t`.[21] The value of a wide-character literal containing a single *c-char* has value equal to the numerical value of the encoding of the *c-char* in the execution wide-character set. The value of a wide-character literal containing multiple *c-char*s is implementation-defined.

3 Certain nongraphic characters, the single quote `'`, the double quote `"`, the question mark `?`, and the backslash `\`, can be represented according to Table 6. The double quote `"` and the question mark `?`, can be represented as themselves or by the escape sequences `\"` and `\?` respectively, but the single quote `'` and the backslash `\` shall be represented by the escape sequences `\'` and `\\` respectively. Escape sequences in which the character following the backslash is not listed in Table 6 are conditionally-supported, with implementation-defined semantics. An escape sequence specifies a single character.

4 The escape *\ooo* consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape *\xhhh* consists of the backslash followed by `x` followed by one or more hexadecimal digits that are taken to specify the value of the desired character. There is no limit to the number of digits in a hexadecimal sequence. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively. The value of a character

21) They are intended for character sets where a character does not fit into a single byte.

Table 6 — Escape sequences

new-line	NL(LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
alert	BEL	\a
backslash	\	\\
question mark	?	\?
single quote	'	\'
double quote	"	\"
octal number	*ooo*	*ooo*
hex number	*hhh*	\x*hhh*

literal is implementation-defined if it falls outside of the implementation-defined range defined for `char` (for literals with no prefix), `char16_t` (for literals prefixed by `'u'`), `char32_t` (for literals prefixed by `'U'`), or `wchar_t` (for literals prefixed by `'L'`).

5 A universal-character-name is translated to the encoding, in the execution character set, of the character named. If there is no such encoding, the universal-character-name is translated to an implementation-defined encoding. [*Note:* in translation phase 1, a universal-character-name is introduced whenever an actual extended character is encountered in the source text. Therefore, all extended characters are described in terms of universal-character-names. However, the actual compiler implementation may use its own native character set, so long as the same results are obtained. — *end note*]

2.13.3 Floating literals [lex.fcon]

floating-literal:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$. *digit-sequence*
digit-sequence .

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`f l F L`

1 A floating literal consists of an integer part, a decimal point, a fraction part, an `e` or `E`, an optionally signed integer exponent, and an optional type suffix. The integer and fraction parts both consist of a sequence of decimal (base ten) digits. Either the integer part or the fraction part (not both) can be omitted; either the decimal point or the letter `e` (or `E`) and the exponent (not both) can be omitted. The integer part, the optional decimal point and the optional fraction part form the *significant part* of the floating literal. The exponent, if present, indicates the power of 10 by which the significant part is to be scaled. If the scaled

value is in the range of representable values for its type, the result is the scaled value if representable, else the larger or smaller representable value nearest the scaled value, chosen in an implementation-defined manner. The type of a floating literal is `double` unless explicitly specified by a suffix. The suffixes `f` and `F` specify `float`, the suffixes `l` and `L` specify `long double`. If the scaled value is not in the range of representable values for its type, the program is ill-formed.

2.13.4 String literals [lex.string]

string-literal:
" *s-char-sequence*$_{opt}$ "
`u8"` *s-char-sequence*$_{opt}$ "
`u"` *s-char-sequence*$_{opt}$ "
`U"` *s-char-sequence*$_{opt}$ "
`L"` *s-char-sequence*$_{opt}$ "
`R` *raw-string*
`u8R` *raw-string*
`uR` *raw-string*
`UR` *raw-string*
`LR` *raw-string*

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence
universal-character-name

raw-string:
" *d-char-sequence*$_{opt}$ `[` *r-char-sequence*$_{opt}$ `]` *d-char-sequence*$_{opt}$ "

r-char-sequence:
r-char
r-char-sequence r-char

r-char:
any member of the source character set, except
(1), a backslash \followed by a `u` or `U`, or
(2), a right square bracket `]` followed by the initial *d-char-sequence*
(which may be empty) followed by a double quote ".
universal-character-name

d-char-sequence:
d-char
d-char-sequence d-char

d-char:
any member of the basic source character set except:
space, the left square bracket `[`, the right square bracket `]`,
and the control characters representing horizontal tab,
vertical tab, form feed, and newline.

1 A string literal is a sequence of characters (as defined in 2.13.2) surrounded by double quotes, optionally prefixed by `R`, `u8`, `u8R`, `u`, `uR`, `U`, `UR`, `L`, or `LR`, as in `"..."`, `R"..."`, `u8"..."`, `u8R"**[...]**"`, `u"..."`, `uR"*~[...]*~"`, `U"..."`, `UR"zzz[...]zzz"`, `L"..."`, or `LR"[...]"`, respectively.

2 A string literal that has an `R` in the prefix is a *raw string literal*. The terminating *d-char-sequence* of a *raw-string* is the same sequence of characters as the initial *d-char-sequence*. A *d-char-sequence* shall consist of at most 16 characters.

3 [*Note:* A source-file new-line in a raw string literal results in a new-line in the resulting execution *string-literal*, unless preceded by a backslash. Assuming no whitespace at the beginning of lines in the following example, the assert will succeed:

```
const char *p = R"[a
b
c]";
assert(std::strcmp(p, "ab\nc") == 0);
```

— *end note*]

4 A string literal that does not begin with `u8`, `u`, `U`, or `L` is an ordinary string literal, and is initialized with the given characters.

5 A string literal that begins with `u8`, such as `u8"asdf"`, is a UTF-8 string literal and is initialized with the given characters as encoded in UTF-8.

6 Ordinary string literals and UTF-8 string literals are also referred to as narrow string literals. A narrow string literal has type "array of n `const char`", where n is the size of the string as defined below, and has static storage duration (3.7).

7 A string literal that begins with `u`, such as `u"asdf"`, is a `char16_t` string literal. A `char16_t` string literal has type "array of n `const char16_t`", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters. A single *c-char* may produce more than one `char16_t` character in the form of surrogate pairs.

8 A string literal that begins with `U`, such as `U"asdf"`, is a `char32_t` string literal. A `char32_t` string literal has type "array of n `const char32_t`", where n is the size of the string as defined below; it has static storage duration and is initialized with the given characters.

9 A string literal that begins with `L`, such as `L"asdf"`, is a wide string literal. A wide string literal has type "array of n `const wchar_t`", where n is the size of the string as defined belowit has static storage duration and is initialized with the given characters.

10 Whether all string literals are distinct (that is, are stored in nonoverlapping objects) is implementation-defined. The effect of attempting to modify a string literal is undefined.

11 In translation phase 6 (2.1), adjacent string literals are concatenated. If both string literals have the same prefix, the resulting concatenated string literal has that prefix. If one string literal has no prefix, it is treated as a string literal of the same prefix as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally supported with implementation-defined behavior. [*Note:* This concatenation is an interpretation, not a conversion. — *end note*] [*Example:* Here are some examples of valid concatenations:

Table 7 — String literal concatenations

Source		Means	Source		Means	Source		Means
`u"a"`	`u"b"`	`u"ab"`	`U"a"`	`U"b"`	`U"ab"`	`L"a"`	`L"b"`	`L"ab"`
`u"a"`	`"b"`	`u"ab"`	`U"a"`	`"b"`	`U"ab"`	`L"a"`	`"b"`	`L"ab"`
`"a"`	`u"b"`	`u"ab"`	`"a"`	`U"b"`	`U"ab"`	`"a"`	`L"b"`	`L"ab"`

— *end example*]

Characters in concatenated strings are kept distinct.

[*Example:*

```
"\xA" "B"
```

contains the two characters `'\xA'` and `'B'` after concatenation (and not the single hexadecimal character `'\xAB'`). — *end example*]

12 After any necessary concatenation, in translation phase 7 (2.1), `'\0'` is appended to every string literal so that programs that scan a string can find its end.

13 Escape sequences in non-raw string literals and universal-character-names in string literals have the same meaning as in character literals (2.13.2), except that the single quote `'` is representable either by itself or by the escape sequence `\'`, and the double quote `"` shall be preceded by a `\`. In a narrow string literal, a universal-character-name may map to more than one `char` element due to *multibyte encoding*. The size of a `char32_t` or wide string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for the terminating `U'\0'` or `L'\0'`. The size of a `char16_t` string literal is the total number of escape sequences, universal-character-names, and other characters, plus one for each character requiring a surrogate pair, plus one for the terminating `u'\0'`. [*Note:* The size of a `char16_t` string literal is the number of code units, not the number of characters. — *end note*] Within `char32_t` and `char16_t` literals, any universal-character-names shall be within the range `0x0` to `0x10FFFF`. The size of a narrow string literal is the total number of escape sequences and other characters, plus at least one for the multibyte encoding of each universal-character-name, plus one for the terminating `'\0'`.

2.13.5 Boolean literals [lex.bool]

boolean-literal:
`false`
`true`

1 The Boolean literals are the keywords `false` and `true`. Such literals have type `bool`. They are not lvalues.

2.13.6 Pointer literals [lex.nullptr]

pointer-literal:
`nullptr`

1 The pointer literal is the keyword `nullptr`. It is an rvalue of type `std::nullptr_t`.

2.13.7 User-defined literals [lex.ext]

user-defined-literal:
user-defined-integer-literal
user-defined-floating-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix

user-defined-floating-literal:
fractional-constant exponent-part$_{opt}$ *ud-suffix*
digit-sequence exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix

user-defined-character-literal:
character-literal ud-suffix

ud-suffix:
identifier

1 If a token matches both *user-defined-literal* and another literal kind, it is treated as the latter. [*Example:* `123_km`, `1.2LL`, `"Hello"s` are all *user-defined-literal*s, but `12LL` is an *integer-literal*. — *end example*]

2 A `user-defined-literal` is treated as a call to a literal operator or literal operator template (13.5.8). To determine the form of this call for a given *user-defined-literal* L with *ud-suffix* X, the *literal-operator-id* whose literal suffix identifier is X is looked up in the context of L using the rules for unqualified name lookup (3.4.1). Let S be the set of declarations found by this lookup. S shall not be empty.

3 If L is a *user-defined-integer-literal*, let n be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `unsigned long long`, the literal L is treated as a call of the form

```
operator "" X(nULL)
```

Otherwise, S shall contain a raw literal operator or a literal operator template (13.5.8) but not both. If S contains a raw literal operator the *literal* L is treated as a call of the form

```
operator "" X("n")
```

Otherwise (S contains a literal operator template), L is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where n is the source character sequence $c_1c_2...c_k$. [*Note:* the sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

4 If L is a *user-defined-floating-literal*, let f be the literal without its *ud-suffix*. If S contains a literal operator with parameter type `long double`, the literal L is treated as a call of the form

```
operator "" X(fL)
```

Otherwise, S shall contain a raw literal operator or a literal operator template (13.5.8) but not both. If S contains a raw literal operator the *literal* L is treated as a call of the form

```
operator "" X("f")
```

Otherwise (S contains a literal operator template), L is treated as a call of the form

```
operator "" X<'c1', 'c2', ... 'ck'>()
```

where f is the source character sequence $c_1c_2...c_k$. [*Note:* the sequence $c_1c_2...c_k$ can only contain characters from the basic source character set. — *end note*]

5 If L is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of characters (or code points) in *str* (i.e., its length excluding the terminating null character). The literal L is treated as a call of the form

```
operator "" X(str, len)
```

6 If L is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. The literal L is treated as a call of the form

```
operator "" X(ch)
```

7 [*Example:*

```
long double operator "" w(long double);
std::string operator "" w(const char16_t*, size_t);
unsigned operator "" w(const char*);
int main() {
  1.2w;       // calls operator "" w(1.2L)
  u"one"w;    // calls operator "" w(u"one", 3)
  12w;        // calls operator "" w("12")
  "two"w;     // error: no applicable literal operator
}
```

— *end example*]

8 In translation phase 6 (2.1), adjacent string literals are concatenated and *user-defined-string-literal*s are considered string literals for that purpose. During concatenation, *ud-suffix*es are removed and ignored and the concatenation process occurs as described in 2.13.4. At the end of phase 6, if a string literal is the result of a concatenation involving at lease one *user-defined-string-literal*, all the participating *user-defined-string-literal*s shall have the same *ud-suffix* and that suffix is applied to the result of the concatenation.

9 [*Example:*

```
int main() {
  L"A" "B" "C"x;  // OK: same as L"ABC"x
  "P"x "Q" "R"y;  // error: two different ud-suffixes
}
```

— *end example*]

3 Basic concepts [basic]

1 [*Note:* this Clause presents the basic concepts of the C++ language. It explains the difference between an *object* and a *name* and how they relate to the notion of an *lvalue*. It introduces the concepts of a *declaration* and a *definition* and presents C++'s notion of *type*, *scope*, *linkage*, and *storage duration*. The mechanisms for starting and terminating a program are discussed. Finally, this Clause presents the *fundamental* types of the language and lists the ways of constructing *compound* types from these. — *end note*]

2 [*Note:* this Clause does not cover concepts that affect only a single part of the language. Such concepts are discussed in the relevant Clauses. — *end note*]

3 An *entity* is a value, object, variable, reference, function, enumerator, type, class member, template, template specialization, namespace, parameter pack, concept, or concept map.

4 A *name* is a use of an *identifier* (2.10), *operator-function-id* (13.5), *conversion-function-id* (12.3.2), or *template-id* (14.2) that denotes an entity or *label* (6.6.4, 6.1).

5 Every name that denotes an entity is introduced by a *declaration*. Every name that denotes a label is introduced either by a `goto` statement (6.6.4) or a *labeled-statement* (6.1).

6 A *variable* is introduced by the declaration of an object. The variable's name denotes the object.

7 Some names denote types, concepts, concept maps, or templates. In general, it is necessary to determine whether a name denotes one of these entities before parsing the program that contains it. The process that determines this is called *name lookup* (3.4).

8 Two names are *the same* if

- — they are *identifier*s composed of the same character sequence, or
- — they are *operator-function-id* s formed with the same operator, or
- — they are *conversion-function-id* s formed with the same type, or
- — they are *template-id* s that refer to the same class or function (14.4), or
- — they are the names of literal operators (13.5.8) formed with the same literal suffix identifier.

9 A name used in more than one translation unit can potentially refer to the same entity in these translation units depending on the linkage (3.5) of the name specified in each translation unit.

3.1 Declarations and definitions [basic.def]

1 A declaration (Clause 7) introduces names into a translation unit or redeclares names introduced by previous declarations. A declaration specifies the interpretation and attributes of these names.

2 A declaration is a *definition* unless it declares a function without specifying the function's body (8.4), it contains the `extern` specifier (7.1.1) or a *linkage-specification*[22] (7.5) and neither an *initializer* nor a *function-body*, it declares a static data member in a class definition (9.4), it is a class name declaration (9.1), it is an *opaque-enum-declaration* (7.2), or it is a `typedef` declaration (7.1.3), a *using-declaration* (7.3.3), or a *using-directive* (7.3.4).

22) Appearing inside the braced-enclosed *declaration-seq* in a *linkage-specification* does not affect whether a declaration is a definition.

[*Example:* all but one of the following are definitions:

```
int a;                          // defines a
extern const int c = 1;         // defines c
int f(int x) { return x+a; }    // defines f and defines x
struct S { int a; int b; };     // defines S, S::a, and S::b
struct X {                      // defines X
  int x;                        // defines non-static data member x
  static int y;                 // declares static data member y
  X(): x(0) { }                 // defines a constructor of X
};
int X::y = 1;                   // defines X::y
enum { up, down };              // defines up and down
namespace N { int d; }          // defines N and N::d
namespace N1 = N;               // defines N1
X anX;                          // defines anX
```

whereas these are just declarations:

```
extern int a;                   // declares a
extern const int c;             // declares c
int f(int);                     // declares f
struct S;                       // declares S
typedef int Int;                // declares Int
extern X anotherX;              // declares anotherX
using N::d;                     // declares N::d
```

— *end example*]

3 [*Note:* in some circumstances, C++ implementations implicitly define the default constructor (12.1), copy constructor (12.8), assignment operator (12.8), or destructor (12.4) member functions. [*Example:* given

```
#include <string>

struct C {
  std::string s;              // std::string is the standard library class (Clause 21)
};

int main() {
  C a;
  C b = a;
  b = a;
}
```

the implementation will implicitly define functions to make the definition of `C` equivalent to

```
struct C {
  std::string s;
  C(): s() { }
  C(const C& x): s(x.s) { }
  C& operator=(const C& x) { s = x.s; return *this; }
  ~C() { }
};
```

— *end example*] — *end note*]

4 [*Note:* a class name can also be implicitly declared by an *elaborated-type-specifier* (7.1.6.3). — *end note*]

5 A program is ill-formed if the definition of any object gives the object an incomplete type (3.9).

3.2 One definition rule [basic.def.odr]

1 No translation unit shall contain more than one definition of any variable, function, class type, concept, concept map, enumeration type, or template.

2 An expression is *potentially evaluated* unless it is an unevaluated operand (Clause 5) or a subexpression thereof. An object or non-overloaded function whose name appears as a potentially-evaluated expression is *used* unless it is an object that satisfies the requirements for appearing in a constant expression (5.19) and the lvalue-to-rvalue conversion (4.1) is immediately applied. A virtual member function is used if it is not pure. An overloaded function is used if it is selected by overload resolution when referred to from a potentially-evaluated expression. [*Note:* this covers calls to named functions (5.2.2), operator overloading (Clause 13), user-defined conversions (12.3.2), allocation function for placement new (5.3.4), as well as non-default initialization (8.5). A copy constructor is used even if the call is actually elided by the implementation. — *end note*] An allocation or deallocation function for a class is used by a new expression appearing in a potentially-evaluated expression as specified in 5.3.4 and 12.5. A deallocation function for a class is used by a delete expression appearing in a potentially-evaluated expression as specified in 5.3.5 and 12.5. A non-placement allocation or deallocation function for a class is used by the definition of a constructor of that class. A non-placement deallocation function for a class is used by the definition of the destructor of that class, or by being selected by the lookup at the point of definition of a virtual destructor (12.4).[23] A copy-assignment function for a class is used by an implicitly-defined copy-assignment function for another class as specified in 12.8. A default constructor for a class is used by default initialization or value initialization as specified in 8.5. A constructor for a class is used as specified in 8.5. A destructor for a class is used as specified in 12.4.

3 Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required. The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined (see 12.1, 12.4 and 12.8). An inline function shall be defined in every translation unit in which it is used.

4 Exactly one definition of a class is required in a translation unit if the class is used in a way that requires the class type to be complete. [*Example:* the following complete translation unit is well-formed, even though it never defines `X`:

```
struct X;                       // declare X as a struct type
struct X* x1;                   // use X in pointer formation
X* x2;                          // use X in pointer formation
```

— *end example*] [*Note:* the rules for declarations and expressions describe in which contexts complete class types are required. A class type `T` must be complete if:

— an object of type `T` is defined (3.1), or

— a non-static class data member of type `T` is declared (9.2), or

— `T` is used as the object type or array element type in a *new-expression* (5.3.4), or

— an lvalue-to-rvalue conversion is applied to an lvalue referring to an object of type `T` (4.1), or

— an expression is converted (either implicitly or explicitly) to type `T` (Clause 4, 5.2.3, 5.2.7, 5.2.9, 5.4), or

23) An implementation is not required to call allocation and deallocation functions from constructors or destructors; however, this is a permissible implementation technique.

— an expression that is not a null pointer constant, and has type other than `void*`, is converted to the type pointer to `T` or reference to `T` using an implicit conversion (Clause 4), a `dynamic_cast` (5.2.7) or a `static_cast` (5.2.9), or

— a class member access operator is applied to an expression of type `T` (5.2.5), or

— the `typeid` operator (5.2.8) or the `sizeof` operator (5.3.3) is applied to an operand of type `T`, or

— a function with a return type or argument type of type `T` is defined (3.1) or called (5.2.2), or

— a class with a base class of type `T` is defined (10), or

— an lvalue of type `T` is assigned to (5.17), or

— the type `T` is the subject of an `alignof` expression (5.3.6).

— *end note*]

5 There can be more than one definition of a class type (Clause 9), concept (14.9), concept map (14.9.2), enumeration type (7.2), inline function with external linkage (7.1.2), class template (Clause 14), non-static function template (14.5.6), static data member of a class template (14.5.1.3), member function of a class template (14.5.1.1), or template specialization for which some template parameters are not specified (14.7, 14.5.5) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements. Given such an entity named `D` defined in more than one translation unit, then

— each definition of `D` shall consist of the same sequence of tokens; and

— in each definition of `D`, corresponding names, looked up according to 3.4, shall refer to an entity defined within the definition of `D`, or shall refer to the same entity, after overload resolution (13.3) and after matching of partial template specialization (14.8.3), except that a name can refer to a `const` object with internal or no linkage if the object has the same literal type in all definitions of `D`, and the object is initialized with a constant expression (5.19), and the value (but not the address) of the object is used, and the object has the same value in all definitions of `D`; and

— in each definition of `D`, the overloaded operators referred to, the implicit calls to conversion functions, constructors, operator new functions and operator delete functions, shall refer to the same function, or to a function defined within the definition of `D`; and

— in each definition of `D`, a default argument used by an (implicit or explicit) function call is treated as if its token sequence were present in the definition of `D`; that is, the default argument is subject to the three requirements described above (and, if the default argument has sub-expressions with default arguments, this requirement applies recursively).[24]

— if `D` is a class with an implicitly-declared constructor (12.1), it is as if the constructor was implicitly defined in every translation unit where it is used, and the implicit definition in every translation unit shall call the same constructor for a base class or a class member of `D`. [*Example:*

```
//translation unit 1:
struct X {
  X(int);
  X(int, int);
};
X::X(int = 0) { }
class D: public X { };
D d2;                           // X(int) called by D()
```

24) 8.3.6 describes how default argument names are looked up.

```
//translation unit 2:
struct X {
  X(int);
  X(int, int);
};
X::X(int = 0, int = 0) { }
class D: public X { };          // X(int, int) called by D();
                                // D()'s implicit definition
                                // violates the ODR
```

— *end example*]

If `D` is a template and is defined in more than one translation unit, then the last four requirements from the list above shall apply to names from the template's enclosing scope used in the template definition (14.6.3), and also to dependent names at the point of instantiation (14.6.2). If the definitions of `D` satisfy all these requirements, then the program shall behave as if there were a single definition of `D`. If the definitions of `D` do not satisfy these requirements, then the behavior is undefined.

3.3 Declarative regions and scopes [basic.scope]

1 Every name is introduced in some portion of program text called a *declarative region*, which is the largest part of the program in which that name is *valid*, that is, in which that name may be used as an unqualified name to refer to the same entity. In general, each particular name is valid only within some possibly discontiguous portion of program text called its *scope*. To determine the scope of a declaration, it is sometimes convenient to refer to the *potential scope* of a declaration. The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.

2 [*Example:* in

```
int j = 24;
int main() {
  int i = j, j;
  j = 42;
}
```

the identifier `j` is declared twice as a name (and used twice). The declarative region of the first `j` includes the entire example. The potential scope of the first `j` begins immediately after that `j` and extends to the end of the program, but its (actual) scope excludes the text between the `,` and the `}`. The declarative region of the second declaration of `j` (the `j` immediately before the semicolon) includes all the text between `{` and `}`, but its potential scope excludes the declaration of `i`. The scope of the second declaration of `j` is the same as its potential scope. — *end example*]

3 The names declared by a declaration are introduced into the scope in which the declaration occurs, except that the presence of a `friend` specifier (11.4), certain uses of the *elaborated-type-specifier* (7.1.6.3), and *using-directives* (7.3.4) alter this general behavior.

4 Given a set of declarations in a single declarative region, each of which specifies the same unqualified name,

— they shall all refer to the same entity, or all refer to functions and function templates; or

— exactly one declaration shall declare a class name or enumeration name that is not a typedef name and the other declarations shall all refer to the same object or enumerator, or all refer to functions and function templates; in this case the class name or enumeration name is hidden (3.3.10). [*Note:* a

namespace name or a class template name must be unique in its declarative region (7.3.2, Clause 14). — *end note*]

[*Note:* these restrictions apply to the declarative region into which a name is introduced, which is not necessarily the same as the region in which the declaration occurs. In particular, *elaborated-type-specifier*s (7.1.6.3) and friend declarations (11.4) may introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to that region. Local extern declarations (3.5) may introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions. — *end note*]

5 [*Note:* the name lookup rules are summarized in 3.4. — *end note*]

3.3.1 Point of declaration [basic.scope.pdecl]

1 The *point of declaration* for a name is immediately after its complete declarator (Clause 8) and before its *initializer* (if any), except as noted below. [*Example:*

```
int x = 12;
{ int x = x; }
```

Here the second `x` is initialized with its own (indeterminate) value. — *end example*]

2 [*Note:* a nonlocal name remains visible up to the point of declaration of the local name that hides it.[*Example:*

```
const int  i = 2;
{ int  i[i]; }
```

declares a local array of two integers. — *end example*] — *end note*]

3 The point of declaration for a class first declared by a *class-specifier* is immediately after the identifier or *simple-template-id* (if any) in its *class-head* (Clause 9). The point of declaration for an enumeration is immediately after the *identifier* (if any) in either its *enum-specifier* (7.2) or its first *opaque-enum-declaration* (7.2), whichever comes first. The point of declaration of a template alias immediately follows the identifier for the alias being declared.

4 The point of declaration for an enumerator is immediately after its *enumerator-definition*.[*Example:*

```
const int x = 12;
{ enum { x = x }; }
```

Here, the enumerator `x` is initialized with the value of the constant `x`, namely 12. — *end example*]

5 After the point of declaration of a class member, the member name can be looked up in the scope of its class. [*Note:* this is true even if the class is an incomplete class. For example,

```
struct X {
  enum E { z = 16 };
  int b[X::z];      // OK
};
```

— *end note*]

6 The point of declaration of a class first declared in an *elaborated-type-specifier* is as follows:

— for a declaraton of the form

class-key identifier attribute-specifier$_{opt}$;

the *identifier* is declared to be a *class-name* in the scope that contains the declaration, otherwise

— for an *elaborated-type-specifier* of the form

class-key identifier

if the *elaborated-type-specifier* is used in the *decl-specifier-seq* or *parameter-declaration-clause* of a function defined in namespace scope, the *identifier* is declared as a *class-name* in the namespace that contains the declaration; otherwise, except as a friend declaration, the *identifier* is declared in the smallest non-class, non-function-prototype scope that contains the declaration. [*Note:* these rules also apply within templates. — *end note*] [*Note:* other forms of *elaborated-type-specifier* do not declare a new name, and therefore must refer to an existing *type-name*. See 3.4.4 and 7.1.6.3. — *end note*]

7 The point of declaration for an *injected-class-name* (9) is immediately following the opening brace of the class definition.

8 The point of declaration for a function-local predefined variable (8.4) is immediately before the *function-body* of a function definition.

9 [*Note:* friend declarations refer to functions or classes that are members of the nearest enclosing namespace, but they do not introduce new names into that namespace (7.3.1.2). Function declarations at block scope and object declarations with the `extern` specifier at block scope refer to declarations that are members of an enclosing namespace, but they do not introduce new names into that scope. — *end note*]

10 The point of declaration for a concept (14.9) is immediately after the identifier in the *concept-definition*. The point of declaration for a concept map (14.9.2) is immediately after the *concept-id* in the *concept-map-definition*.

11 The point of declaration for a template requirement (14.10.1) is immediately after the corresponding *requirement* in a *requires-clause* or the *constrained-template-parameter* in a *template-parameter-list*. The point of declaration of an implied template requirement (14.10.1.2) is the point of declaration of the template requirement from which it was implied or immediately after the declaration of the entity from which the requirement was implied.

12 The point of declaration for the *identifier* in a *concept-instance-alias-def* is immediately after the *concept-id* of its *requirement* or *refinement-specifier*.

13 [*Note:* for point of instantiation of a template, see 14.6.4.1. — *end note*]

3.3.2 Local scope [basic.scope.local]

1 A name declared in a block (6.3) is local to that block. Its potential scope begins at its point of declaration (3.3.1) and ends at the end of its declarative region.

2 The potential scope of a function parameter name (including one appearing in a *lambda-parameter-declaration-clause*) or of a function-local predefined variable in a function definition (8.4) begins at its point of declaration. If the function has a *function-try-block* the potential scope of a parameter or of a function-local predefined variable ends at the end of the last associated handler, otherwise it ends at the end of the outermost block of the function definition. A parameter name shall not be redeclared in the outermost block of the function definition nor in the outermost block of any handler associated with a *function-try-block*.

3 The name in a `catch` exception-declaration is local to the handler and shall not be redeclared in the outermost block of the handler.

4 Names declared in the *for-init-statement*, the *for-range-declaration*, and in the *condition* of `if`, `while`, `for`, and `switch` statements are local to the `if`, `while`, `for`, or `switch` statement (including the controlled

statement), and shall not be redeclared in a subsequent condition of that statement nor in the outermost block (or, for the `if` statement, any of the outermost blocks) of the controlled statement; see 6.4.

3.3.3 Function prototype scope [basic.scope.proto]

1 In a function declaration, or in any function declarator except the declarator of a function definition (8.4), names of parameters (if supplied) have function prototype scope, which terminates at the end of the nearest enclosing function declarator.

3.3.4 Function scope [basic.funscope]

1 Labels (6.1) have *function scope* and may be used anywhere in the function in which they are declared. Only labels have function scope.

3.3.5 Namespace scope [basic.scope.namespace]

1 The declarative region of a *namespace-definition* is its *namespace-body*. The potential scope denoted by an *original-namespace-name* is the concatenation of the declarative regions established by each of the *namespace-definitions* in the same declarative region with that *original-namespace-name*. Entities declared in a *namespace-body* are said to be *members* of the namespace, and names introduced by these declarations into the declarative region of the namespace are said to be *member names* of the namespace. A namespace member name has namespace scope. Its potential scope includes its namespace from the name's point of declaration (3.3.1) onwards; and for each *using-directive* (7.3.4) that nominates the member's namespace, the member's potential scope includes that portion of the potential scope of the *using-directive* that follows the member's point of declaration. [*Example:*

```
namespace N {
  int i;
  int g(int a) { return a; }
  int j();
  void q();
}
namespace { int l=1; }
// the potential scope of l is from its point of declaration
// to the end of the translation unit

namespace N {
  int g(char a) {   // overloads N::g(int)
    return l+a;     // l is from unnamed namespace
  }

  int i;            // error: duplicate definition
  int j();          // OK: duplicate function declaration

  int j() {         // OK: definition of N::j()
    return g(i);    // calls N::g(int)
  }
  int q();          // error: different return type
}
```

— *end example*]

2 A namespace member can also be referred to after the `::` scope resolution operator (5.1) applied to the name of its namespace or the name of a namespace which nominates the member's namespace in a *using-directive;* see 3.4.3.2.

3 The outermost declarative region of a translation unit is also a namespace, called the *global namespace*. A name declared in the global namespace has *global namespace scope* (also called *global scope*). The potential scope of such a name begins at its point of declaration (3.3.1) and ends at the end of the translation unit that is its declarative region. Names with global namespace scope are said to be *global*.

3.3.6 Class scope [basic.scope.class]

1 The following rules describe the scope of names declared in classes.

1) The potential scope of a name declared in a class consists not only of the declarative region following the name's point of declaration, but also of all function bodies, *brace-or-equal-initializers* of non-static data members, and default arguments in that class (including such things in nested classes).

2) A name `N` used in a class `S` shall refer to the same declaration in its context and when re-evaluated in the completed scope of `S`. No diagnostic is required for a violation of this rule.

3) If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program is ill-formed, no diagnostic is required.

4) A name declared within a member function hides a declaration of the same name whose scope extends to or past the end of the member function's class.

5) The potential scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if the members are defined lexically outside the class (this includes static data member definitions, nested class definitions, member function definitions (including the member function body and any portion of the declarator part of such definitions which follows the *declarator-id*, including a *parameter-declaration-clause* and any default arguments (8.3.6).[*Example:*

```
typedef int  c;
enum { i = 1 };

class X {
  char  v[i];                       // error: i refers to ::i
                                    // but when reevaluated is X::i
  int  f() { return sizeof(c); }    // OK: X::c
  char  c;
  enum { i = 2 };
};

typedef char*  T;
struct Y {
  T  a;                             // error: T refers to ::T
                                    // but when reevaluated is Y::T
  typedef long  T;
  T  b;
};

typedef int I;
class D {
  typedef I I;                      // error, even though no reordering involved
};
```

— *end example*]

2 The name of a class member shall only be used as follows:

— in the scope of its class (as described above) or a class derived (Clause 10) from its class,

— after the `.` operator applied to an expression of the type of its class (5.2.5) or a class derived from its class,

— after the `->` operator applied to a pointer to an object of its class (5.2.5) or a class derived from its class,

— after the `::` scope resolution operator (5.1) applied to the name of its class or a class derived from its class.

3.3.7 Concept scope [basic.scope.concept]

1 The following rules describe the scope of names declared in concepts and concept maps.

1) The potential scope of a name declared in a concept or concept map consists not only of the declarative region following the name's point of declaration, but also of all associated function bodies in that concept or concept map.

2) A name `N` used in a concept or concept map `S` shall refer to the same declaration in its context and when re-evaluated in the completed scope of `S`. No diagnostic is required for a violation of this rule.

3) If reordering declarations in a concept or concept map yields an alternate valid program under (1), the program is ill-formed, no diagnostic is required.

4) A name declared within an associated function definition hides a declaration of the same name whose scope extends to or past the end of the associated function's concept or concept map.

3.3.8 Requirements scope [basic.scope.req]

1 A template requirement has *requirements scope*. Its potential scope begins immediately after its *concept-id* and terminates at the end of the constrained template (14.10) or constrained member (9.2).

2 In a constrained context (14.10), the names of all associated functions inside the concepts named by the concept requirements in the template's requirements are declared in the same scope as the constrained template's template parameters. Each of these names refers to one or more members of the concept map archetypes (14.10.2) that correspond to the concept requirements. [*Note:* The declaration of these names in the scope of the template parameters does not establish archetypes unless name lookup finds these declarations. — *end note*] [*Example:*

```
concept A<class B> {
 void g(const B&);
}

template<class T, class U>
requires A<U>
void f(T & x, U & y) {
   g(y);       // binds to A<U'>'::g(const U' &)
   g(x);       // error: no overload of g takes T' values.
}
```

— *end example*]

3.3.9 Enumeration scope [basic.scope.enum]

1 The name of a scoped enumerator (7.2) has *enumeration scope*. Its potential scope begins at its point of declaration and terminates at the end of the *enum-specifier*.

3.3.10 Name hiding [basic.scope.hiding]

1 A name can be hidden by an explicit declaration of that same name in a nested declarative region, more refined concept (14.9.3), or derived class (10.2).

2 A class name (9.1) or enumeration name (7.2) can be hidden by the name of an object, function, or enumerator declared in the same scope. If a class or enumeration name and an object, function, or enumerator are declared in the same scope (in any order) with the same name, the class or enumeration name is hidden wherever the object, function, or enumerator name is visible.

3 In a member function definition, the declaration of a local name hides the declaration of a member of the class with the same name; see 3.3.6. The declaration of a member in a derived class (Clause 10) hides the declaration of a member of a base class of the same name; see 10.2.

4 During the lookup of a name qualified by a namespace name, declarations that would otherwise be made visible by a *using-directive* can be hidden by declarations with the same name in the namespace containing the *using-directive;* see (3.4.3.2).

5 If a name is in scope and is not hidden it is said to be *visible*.

6 In an associated function definition, the declaration of a local name hides the declaration of a member of the concept or concept map with the same name; see 3.3.7.

3.4 Name lookup [basic.lookup]

1 The name lookup rules apply uniformly to all names (including *typedef-name*s (7.1.3), *namespace-name*s (7.3), *concept-name*s (14.9), *concept-map-name*s (14.9.2), and *class-name*s (9.1)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a declaration (3.1) of that name. Name lookup shall find an unambiguous declaration for the name (see 10.2). Name lookup may associate more than one declaration with a name if it finds the name to be a function name; the declarations are said to form a set of overloaded functions (13.1). Overload resolution (13.3) takes place after name lookup has succeeded. The access rules (Clause 11) are considered only once name lookup and function overload resolution (if applicable) have succeeded. Only after name lookup, function overload resolution (if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (Clause 5).

2 A name "looked up in the context of an expression" is looked up as an unqualified name in the scope where the expression is found.

3 The injected-class-name of a class (Clause 9) is also considered to be a member of that class for the purposes of name hiding and lookup.

4 [*Note:* 3.5 discusses linkage issues. The notions of scope, point of declaration and name hiding are discussed in 3.3. — *end note*]

3.4.1 Unqualified name lookup [basic.lookup.unqual]

1 In all the cases listed in 3.4.1, the scopes are searched for a declaration in the order listed in each of the respective categories; name lookup ends as soon as a declaration is found for the name. If no declaration is found, the program is ill-formed.

2 The declarations from the namespace nominated by a *using-directive* become visible in a namespace enclosing the *using-directive*; see 7.3.4. For the purpose of the unqualified name lookup rules described in 3.4.1, the declarations from the namespace nominated by the *using-directive* are considered members of that enclosing namespace.

3 The lookup for an unqualified name used as the *postfix-expression* of a function call is described in 3.4.2. [*Note:* for purposes of determining (during parsing) whether an expression is a *postfix-expression* for a function call, the usual name lookup rules apply. The rules in 3.4.2 have no effect on the syntactic interpretation of an expression. For example,

```
typedef int f;
namespace N {
  struct A {
    friend void f(A &);
    operator int();
    void g(A a) {
      int i = f(a);          // f is the typedef, not the friend
                             // function: equivalent to int(a)
    }
  };
}
```

Because the expression is not a function call, the argument-dependent name lookup (3.4.2) does not apply and the friend function `f` is not found. — *end note*]

4 A name used in global scope, outside of any function, class or user-declared namespace, shall be declared before its use in global scope.

5 A name used in a user-declared namespace outside of the definition of any function or class shall be declared before its use in that namespace or before its use in a namespace enclosing its namespace.

6 A name used in the definition of a function following the function's *declarator-id*[25] that is a member of namespace `N` (where, only for the purpose of exposition, `N` could represent the global scope) shall be declared before its use in the block in which it is used or in one of its enclosing blocks (6.3) or, shall be declared before its use in namespace `N` or, if `N` is a nested namespace, shall be declared before its use in one of `N`'s enclosing namespaces. [*Example:*

```
namespace A {
  namespace N {
    void f();
  }
}
void A::N::f() {
  i = 5;
  // The following scopes are searched for a declaration of i:
  // 1) outermost block scope of A::N::f, before the use of i
  // 2) scope of namespace N
  // 3) scope of namespace A
  // 4) global scope, before the definition of A::N::f
}
```

— *end example*]

25) This refers to unqualified names that occur, for instance, in a type or default argument expression in the *parameter-declaration-clause* or used in the function body.

7 A name used in the definition of a class X outside of a member function body or nested class definition[26] shall be declared in one of the following ways:

— before its use in class X or be a member of a base class of X (10.2), or

— if X is a nested class of class Y (9.7), before the definition of X in Y, or shall be a member of a base class of Y (this lookup applies in turn to Y 's enclosing classes, starting with the innermost enclosing class),[27] or

— if X is a local class (9.8) or is a nested class of a local class, before the definition of class X in a block enclosing the definition of class X, or

— if X is a member of namespace N, or is a nested class of a class that is a member of N, or is a local class or a nested class within a local class of a function that is a member of N, before the definition of class X in namespace N or in one of N 's enclosing namespaces.

[*Example:*

```
namespace M {
  class B { };
}

namespace N {
  class Y : public M::B {
    class X {
      int a[i];
    };
  };
}

// The following scopes are searched for a declaration of i:
// 1) scope of class N::Y::X, before the use of i
// 2) scope of class N::Y, before the definition of N::Y::X
// 3) scope of N::Y's base class M::B
// 4) scope of namespace N, before the definition of N::Y
// 5) global scope, before the definition of N
```

— *end example*] [*Note:* when looking for a prior declaration of a class or function introduced by a `friend` declaration, scopes outside of the innermost enclosing namespace scope are not considered; see 7.3.1.2. — *end note*] [*Note:* 3.3.6 further describes the restrictions on the use of names in a class definition. 9.7 further describes the restrictions on the use of names in nested class definitions. 9.8 further describes the restrictions on the use of names in local class definitions. — *end note*]

8 A name used in the definition of a member function (9.3) of class X following the function's *declarator-id* [28] or in the *brace-or-equal-initializer* of a non-static data member (9.2) of class X shall be declared in one of the following ways:

— before its use in the block in which it is used or in an enclosing block (6.3), or

— shall be a member of class X or be a member of a base class of X (10.2), or

26) This refers to unqualified names following the class name; such a name may be used in the *base-clause* or may be used in the class definition.

27) This lookup applies whether the definition of X is nested within Y's definition or whether X's definition appears in a namespace scope enclosing Y 's definition (9.7).

28) That is, an unqualified name that occurs, for instance, in a type or default argument expression in the *parameter-declaration-clause* or in the function body.

— if X is a nested class of class Y (9.7), shall be a member of Y, or shall be a member of a base class of Y (this lookup applies in turn to Y's enclosing classes, starting with the innermost enclosing class),[29] or

— if X is a local class (9.8) or is a nested class of a local class, before the definition of class X in a block enclosing the definition of class X, or

— if X is a member of namespace N, or is a nested class of a class that is a member of N, or is a local class or a nested class within a local class of a function that is a member of N, before the use of the name, in namespace N or in one of N 's enclosing namespaces.

[*Example:*

```
class B { };
namespace M {
  namespace N {
    class X : public B {
      void f();
    };
  }
}
void M::N::X::f() {
  i = 16;
}

// The following scopes are searched for a declaration of i:
// 1) outermost block scope of M::N::X::f, before the use of i
// 2) scope of class M::N::X
// 3) scope of M::N::X's base class B
// 4) scope of namespace M::N
// 5) scope of namespace M
// 6) global scope, before the definition of M::N::X::f
```

— *end example*] [*Note:* 9.3 and 9.4 further describe the restrictions on the use of names in member function definitions. 9.7 further describes the restrictions on the use of names in the scope of nested classes. 9.8 further describes the restrictions on the use of names in local class definitions. — *end note*]

9 Name lookup for a name used in the definition of a `friend` function (11.4) defined inline in the class granting friendship shall proceed as described for lookup in member function definitions. If the `friend` function is not defined in the class granting friendship, name lookup in the `friend` function definition shall proceed as described for lookup in namespace member function definitions.

10 In a `friend` declaration naming a member function, a name used in the function declarator and not part of a *template-argument* in a *template-id* is first looked up in the scope of the member function's class. If it is not found, or if the name is part of a *template-argument* in a *template-id*, the look up is as described for unqualified names in the definition of the class granting friendship. [*Example:*

```
struct A {
  typedef int AT;
  void f1(AT);
  void f2(float);
};
struct B {
  typedef float BT;
  friend void A::f1(AT);      // parameter type is A::AT
```

29) This lookup applies whether the member function is defined within the definition of class X or whether the member function is defined in a namespace scope enclosing X's definition.

```
  friend void A::f2(BT);       // parameter type is B::BT
};
```

— *end example*]

11 During the lookup for a name used as a default argument (8.3.6) in a function *parameter-declaration-clause* or used in the *expression* of a *mem-initializer* for a constructor (12.6.2), the function parameter names are visible and hide the names of entities declared in the block, class or namespace scopes containing the function declaration. [*Note:* 8.3.6 further describes the restrictions on the use of names in default arguments. 12.6.2 further describes the restrictions on the use of names in a *ctor-initializer*. — *end note*]

12 During the lookup of a name used in the *constant-expression* of an *enumerator-definition*, previously declared *enumerator*s of the enumeration are visible and hide the names of entities declared in the block, class, or namespace scopes containing the *enum-specifier*.

13 A name used in the definition of a `static` data member of class `X` (9.4.2) (after the *qualified-id* of the static member) is looked up as if the name was used in a member function of `X`. [*Note:* 9.4.2 further describes the restrictions on the use of names in the definition of a `static` data member. — *end note*]

14 If a variable member of a namespace is defined outside of the scope of its namespace then any name used in the definition of the variable member (after the *declarator-id*) is looked up as if the definition of the variable member occurred in its namespace. [*Example:*

```
namespace N {
  int i = 4;
  extern int j;
}

int i = 2;

int N::j = i;                  // N::j == 4
```

— *end example*]

15 A name used in the handler for a *function-try-block* (Clause 15) is looked up as if the name was used in the outermost block of the function definition. In particular, the function parameter names shall not be redeclared in the *exception-declaration* nor in the outermost block of a handler for the *function-try-block*. Names declared in the outermost block of the function definition are not found when looked up in the scope of a handler for the *function-try-block*. [*Note:* but function parameter names are found. — *end note*]

16 A name used in the definition of a concept or concept map `X` outside of an associated function body shall be declared in one of the following ways:

— before its use in the concept or concept map `X` or be a member of a less refined concept of `X`, or

— if `X` is a member of namespace `N`, before the definition of concept or concept map `X` in namespace `N` or in one of `N`'s enclosing namespaces.

[*Example:*

```
typedef int result_type;
concept C<class F, class T1> {
  result_type operator() (F&, T1);
  typename result_type; // error result_type used before declared
}
```

— *end example*]

17 A name used in the definition of an associated function (14.9.1.1) of a concept or concept map `X` following the associated function's *declarator-id* shall be declared in one of the following ways:

- before its use in the block in which it is used or in an enclosing block (6.3), or
- shall be a member of concept or concept map `X` or be a member of a less refined concept of `X`, or
- if `X` is a member of namespace `N`, before the associated function definition, in namespace `N` or in one of `N`'s enclosing namespaces.

18 [*Note:* the rules for name lookup in template definitions are described in 14.6. — *end note*]

3.4.2 Argument-dependent name lookup [basic.lookup.argdep]

1 When an unqualified name is used as the *postfix-expression* in a function call (5.2.2), other namespaces not considered during the usual unqualified lookup (3.4.1) may be searched, and in those namespaces, namespace-scope friend function declarations (11.4) not otherwise visible may be found. These modifications to the search depend on the types of the arguments (and for template template arguments, the namespace of the template argument).

2 For each argument type `T` in the function call, there is a set of zero or more associated namespaces and a set of zero or more associated classes to be considered. The sets of namespaces and classes is determined entirely by the types of the function arguments (and the namespace of any template template argument). Typedef names and *using-declaration*s used to specify the types do not contribute to this set. The sets of namespaces and classes are determined in the following way:

- If `T` is a fundamental type, its associated sets of namespaces and classes are both empty.
- If `T` is a non-archetype class type (including unions), its associated classes are: the class itself; the class of which it is a member, if any; and its direct and indirect base classes. Its associated namespaces are the namespaces of which its associated classes are members. Furthermore, if `T` is a class template specialization, its associated namespaces and classes also include: the namespaces and classes associated with the types of the template arguments provided for template type parameters (excluding template template parameters); the namespaces of which any template template arguments are members; and the classes of which any member templates used as template template arguments are members. [*Note:* non-type template arguments do not contribute to the set of associated namespaces. — *end note*]
- If `T` is an enumeration type, its associated namespace is the namespace in which it is defined. If it is class member, its associated class is the member's class; else it has no associated class.
- If `T` is a pointer to `U` or an array of `U`, its associated namespaces and classes are those associated with `U`.
- If `T` is a function type, its associated namespaces and classes are those associated with the function parameter types and those associated with the return type.
- If `T` is a pointer to a member function of a class `X`, its associated namespaces and classes are those associated with the function parameter types and return type, together with those associated with `X`.
- If `T` is a pointer to a data member of class `X`, its associated namespaces and classes are those associated with the member type together with those associated with `X`.

If an associated namespace is an inline namespace (7.3.1), its enclosing namespace is also included in the set. If an associated namespace directly contains inline namespaces, those inline namespaces are also included in the set. In addition, if the argument is the name or address of a set of overloaded functions and/or function templates, its associated classes and namespaces are the union of those associated with each of the

members of the set: the namespace in which the function or function template is defined and the classes and namespaces associated with its (non-dependent) parameter types and return type.

3 Let X be the lookup set produced by unqualified lookup (3.4.1) and let Y be the lookup set produced by argument dependent lookup (defined as follows). If X contains

— a declaration of a class member, or

— a block-scope function declaration that is not a *using-declaration*, or

— a declaration that is neither a function or a function template

then Y is empty. Otherwise Y is the set of declarations found in the namespaces associated with the argument types as described below. The set of declarations found by the lookup of the name is the union of X and Y. [*Note:* the namespaces and classes associated with the argument types can include namespaces and classes already considered by the ordinary unqualified lookup. — *end note*] [*Example:*

```
namespace NS {
  class T { };
  void f(T);
  void g(T, int);
}
NS::T parm;
void g(NS::T, float);
int main() {
  f(parm);                      // OK: calls NS::f
  extern void g(NS::T, float);
  g(parm, 1);                   // OK: calls g(NS::T, float)
}
```

— *end example*]

4 When considering an associated namespace, the lookup is the same as the lookup performed when the associated namespace is used as a qualifier (3.4.3.2) except that:

— Any *using-directive*s in the associated namespace are ignored.

— Any namespace-scope friend functions or friend function templates declared in associated classes are visible within their respective namespaces even if they are not visible during an ordinary lookup (11.4).

— All names except those of (possibly overloaded) functions and function templates are ignored.

3.4.3 Qualified name lookup [basic.lookup.qual]

1 The name of a class, concept map (but not a concept), or namespace member or enumerator can be referred to after the `::` scope resolution operator (5.1) applied to a *nested-name-specifier* that nominates its class, concept map, namespace, or enumeration. During the lookup for a name preceding the `::` scope resolution operator, object, function, and enumerator names are ignored. If the name found does not designate a namespace, concept map, or a class, enumeration, or dependent type, the program is ill-formed.[*Example:*

```
class A {
public:
  static int n;
};
int main() {
  int A;
  A::n = 42;        // OK
  A b;              // ill-formed: A does not name a type
```

```
}
```

— *end example*]

2 [*Note:* multiply qualified names, such as `N1::N2::N3::n`, can be used to refer to members of nested classes (9.7) or members of nested namespaces. — *end note*]

3 In a declaration in which the *declarator-id* is a *qualified-id*, names used before the *qualified-id* being declared are looked up in the defining namespace scope; names following the *qualified-id* are looked up in the scope of the member's class or namespace. [*Example:*

```
class X { };
class C {
  class X { };
  static const int number = 50;
  static X arr[number];
};
X C::arr[number];    // ill-formed:
                     // equivalent to: ::X C::arr[C::number];
                     // not to: C::X C::arr[C::number];
```

— *end example*]

4 A name prefixed by the unary scope operator `::` (5.1) is looked up in global scope, in the translation unit where it is used. The name shall be declared in global namespace scope or shall be a name whose declaration is visible in global scope because of a *using-directive* (3.4.3.2). The use of `::` allows a global name to be referred to even if its identifier has been hidden (3.3.10).

5 A name prefixed by a *nested-name-specifier* that nominates an enumeration type shall represent an *enumerator* of that enumeration.

6 If a *pseudo-destructor-name* (5.2.4) contains a *nested-name-specifier*, the *type-name*s are looked up as types in the scope designated by the *nested-name-specifier*. Similarly, in a *qualified-id* of the form:

::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* `::` $\sim$ *class-name*

the second *class-name* is looked up in the same scope as the first. [*Example:*

```
struct C {
  typedef int I;
};
typedef int I1, I2;
extern int* p;
extern int* q;
p->C::I::~I();       // I is looked up in the scope of C
q->I1::~I2();        // I2 is looked up in the scope of
                     // the postfix-expression

struct A {
  ~A();
};
typedef A AB;
int main() {
  AB *p;
  p->AB::~AB();      // explicitly calls the destructor for A
}
```

— *end example*] [*Note:* 3.4.5 describes how name lookup proceeds after the `.` and `->` operators. — *end note*]

7 In a constrained context (14.10), a name prefixed by a *nested-name-specifier* that nominates a template type parameter `T` is looked up as follows: for each template requirement `C<`*args*`>` whose template argument list references T, the name is looked up as if the *nested-name-specifier* referenced `C<`*args*`>` instead of T (3.4.3.3), except that only the names of associated types are visible during this lookup. If an associated type of at least one requirement is found, then each name found shall refer to the same type. Otherwise, if the reference to the name occurs within a constrained context, the name is looked up within the scope of the archetype associated with T (and no special restriction on name visibility is in effect for this lookup). [*Example:*

```
concept C<typename T> {
  typename assoc_type;
}

template<typename T, typename U> requires C<T> && C<U>
  T::assoc_type     // okay: refers to C<T>::assoc_type
  f();
```

— *end example*]

3.4.3.1 Class members [class.qual]

1 If the *nested-name-specifier* of a *qualified-id* nominates a class, the name specified after the *nested-name-specifier* is looked up in the scope of the class (10.2), except for the cases listed below. The name shall represent one or more members of that class or of one of its base classes (Clause 10). [*Note:* a class member can be referred to using a *qualified-id* at any point in its potential scope (3.3.6). — *end note*] The exceptions to the name lookup rule above are the following:

— a destructor name is looked up as specified in 3.4.3;

— a *conversion-type-id* of an *conversion-function-id* is looked up both in the scope of the class and in the context in which the entire *postfix-expression* occurs and shall refer to the same type in both contexts;

— the names in a *template-argument* of a *template-id* are looked up in the context in which the entire *postfix-expression* occurs.

— the lookup for a name specified in a *using-declaration* (7.3.3) also finds class or enumeration names hidden within the same scope (3.3.10).

2 In a lookup in which the constructor is an acceptable lookup result and the *nested-name-specifier* nominates a class `C`:

— if the name specified after the *nested-name-specifier*, when looked up in `C`, is the injected-class-name of `C` (Clause 9), or

— if the name specified after the *nested-name-specifier* is the same as the *identifier* or the *simple-template-id*'s *template-name* in the last component of the *nested-name-specifier*,

the name is instead considered to name the constructor of class `C`. [*Note:* for example, the constructor is not an acceptable lookup result in an *elaborated-type-specifier* so the constructor would not be used in place of the injected-class-name. — *end note*] Such a constructor name shall be used only in the *declarator-id* of a declaration that names a constructor or in a *using-declaration*. [*Example:*

```
struct A { A(); };
struct B: public A { B(); };

A::A() { }
```

```
B::B() { }

B::A ba;            // object of type A
A::A a;             // error, A::A is not a type name
struct A::A a2;     // object of type A
```

— *end example*]

3 A class member name hidden by a name in a nested declarative region or by the name of a derived class member can still be found if qualified by the name of its class followed by the `::` operator.

3.4.3.2 Namespace members [namespace.qual]

1 If the *nested-name-specifier* of a *qualified-id* nominates a namespace, the name specified after the *nested-name-specifier* is looked up in the scope of the namespace, except that the names in a *template-argument* of a *template-id* are looked up in the context in which the entire *postfix-expression* occurs.

2 Given `X::m` (where `X` is a user-declared namespace), or given `::m` (where X is the global namespace), let `S` be the set of all declarations of `m` in `X` and in the transitive closure of all namespaces nominated by *using-directive*s in `X` and its used namespaces, except that *using-directive*s that nominate non-inline namespaces (7.3.1) are ignored in any namespace, including `X`, directly containing one or more declarations of `m`. No namespace is searched more than once in the lookup of a name. If `S` is the empty set, the program is ill-formed. Otherwise, if `S` has exactly one member, or if the context of the reference is a *using-declaration* (7.3.3), `S` is the required set of declarations of `m`. Otherwise if the use of `m` is not one that allows a unique declaration to be chosen from `S`, the program is ill-formed. [*Example:*

```
int x;
namespace Y {
  void f(float);
  void h(int);
}

namespace Z {
  void h(double);
}

namespace A {
  using namespace Y;
  void f(int);
  void g(int);
  int i;
}

namespace B {
  using namespace Z;
  void f(char);
  int i;
}

namespace AB {
  using namespace A;
  using namespace B;
  void g();
}
```

```
void h()
{
  AB::g();          // g is declared directly in AB,
                    // therefore S is { AB::g() } and AB::g() is chosen
  AB::f(1);         // f is not declared directly in AB so the rules are
                    // applied recursively to A and B;
                    // namespace Y is not searched and Y::f(float)
                    // is not considered;
                    // S is { A::f(int), B::f(char) } and overload
                    // resolution chooses A::f(int)
  AB::f('c');       // as above but resolution chooses B::f(char)

  AB::x++;          // x is not declared directly in AB, and
                    // is not declared in A or B , so the rules are
                    // applied recursively to Y and Z,
                    // S is { } so the program is ill-formed
  AB::i++;          // i is not declared directly in AB so the rules are
                    // applied recursively to A and B,
                    // S is { A::i , B::i } so the use is ambiguous
                    // and the program is ill-formed
  AB::h(16.8);      // h is not declared directly in AB and
                    // not declared directly in A or B so the rules are
                    // applied recursively to Y and Z,
                    // S is { Y::h(int), Z::h(double) } and overload
                    // resolution chooses Z::h(double)
}
```

3 The same declaration found more than once is not an ambiguity (because it is still a unique declaration). For example:

```
namespace A {
  int a;
}

namespace B {
  using namespace A;
}

namespace C {
  using namespace A;
}

namespace BC {
  using namespace B;
  using namespace C;
}

void f()
{
  BC::a++;          // OK: S is { A::a, A::a }
}

namespace D {
  using A::a;
}
```

```
namespace BD {
  using namespace B;
  using namespace D;
}

void g()
{
  BD::a++;          // OK: S is {  A::a,  A::a }
}
```

4 Because each referenced namespace is searched at most once, the following is well-defined:

```
namespace B {
  int b;
}

namespace A {
  using namespace B;
  int a;
}

namespace B {
  using namespace A;
}

void f()
{
  A::a++;           // OK: a declared directly in A, S is {A::a}
  B::a++;           // OK: both A and B searched (once), S is {A::a}
  A::b++;           // OK: both A and B searched (once), S is {B::b}
  B::b++;           // OK: b declared directly in B, S is {B::b}
}
```

— *end example*]

5 During the lookup of a qualified namespace member name, if the lookup finds more than one declaration of the member, and if one declaration introduces a class name or enumeration name and the other declarations either introduce the same object, the same enumerator or a set of functions, the non-type name hides the class or enumeration name if and only if the declarations are from the same namespace; otherwise (the declarations are from different namespaces), the program is ill-formed. [*Example:*

```
namespace A {
  struct x { };
  int x;
  int y;
}

namespace B {
  struct y { };
}

namespace C {
  using namespace A;
  using namespace B;
  int i = C::x;     // OK, A::x (of type int )
  int j = C::y;     // ambiguous, A::y or B::y
```

```
}
```

— *end example*]

6 In a declaration for a namespace member in which the *declarator-id* is a *qualified-id*, given that the *qualified-id* for the namespace member has the form

nested-name-specifier unqualified-id

the *unqualified-id* shall name a member of the namespace designated by the *nested-name-specifier*. [*Example:*

```
namespace A {
  namespace B {
    void f1(int);
  }
  using namespace B;
}
void A::f1(int){ }  // ill-formed, f1 is not a member of A
```

— *end example*] However, in such namespace member declarations, the *nested-name-specifier* may rely on *using-directive*s to implicitly provide the initial part of the *nested-name-specifier*. [*Example:*

```
namespace A {
  namespace B {
    void f1(int);
  }
}

namespace C {
  namespace D {
    void f1(int);
  }
}

using namespace A;
using namespace C::D;
void B::f1(int){ }  // OK, defines A::B::f1(int)
```

— *end example*]

3.4.3.3 Concept map members [concept.qual]

1 If the *nested-name-specifier* of a *qualified-id* nominates a concept instance, the name specified after the *nested-name-specifier* is looked up as follows:

— If the template argument list of the concept instance depends on a template parameter, and if the name, when looked up within the scope of the concept (not the concept instance), refers to an associated type or class template, the result of name lookup is a dependent type composed of the concept instance and the associated type or class template. [*Note:* this implies that, given two distinct type parameters `T` and `U`, `C<T>::type` and `C<U>::type` are distinct types (although they may alias the same archetype). Also, lookup of 'type' within `C<T>` and `C<U>` does not require the creation of a concept map archetype for `C<T>` or `C<U>`. — *end note*]

— Otherwise, concept map lookup (14.10.1.1) first determines which concept map is referred to by the *nested-name-specifier*. Then concept member lookup (14.9.3.1) is used to find the name within the scope of the concept map. The name shall represent one or more members of that concept map or the concept maps corresponding to the concept's less refined concepts. [*Note:* this lookup requires

a concept map definition, so if the template argument list of the concept instance is dependent on a template parameter (and, therefore, the name does not refer to an associated type or class template—a case which would have been handled by the previous bullet), a concept map archetype definition is required. — *end note*]

[*Note:* Outside of a constrained context, this means that one or more requirement members (14.9.2) will be found, and since those names are synonyms for sets of other names, the result of name lookup is the union of each of those sets. — *end note*] . [*Note:* a concept map member can be referred to using a *qualified-id* at any point in its potential scope (3.3.7). [*Example:*

```
concept C<typename F, typename T1> {
  typename type;
  type operator()(F&, T1);
}

template<typename F, typename T1>
requires C<F, T1>
C<F, T1>::result_type g(F& f, const T1& t1) {
  return f(t1);
}
```

— *end example*] — *end note*]

2 A concept map member name hidden by a name in a nested declarative region or by the name of a more refined concept's member can still be found if qualified by the name of its concept map followed by the `::` operator.

3.4.4 Elaborated type specifiers [basic.lookup.elab]

1 An *elaborated-type-specifier* (7.1.6.3) may be used to refer to a previously declared *class-name* or *enum-name* even though the name has been hidden by a non-type declaration (3.3.10).

2 If the *elaborated-type-specifier* has no *nested-name-specifier*, and unless the *elaborated-type-specifier* appears in a declaration with the following form:

class-key identifier attribute-specifier$_{opt}$;

the *identifier* is looked up according to 3.4.1 but ignoring any non-type names that have been declared. If the *elaborated-type-specifier* is introduced by the `enum` keyword and this lookup does not find a previously declared *type-name*, the *elaborated-type-specifier* is ill-formed. If the *elaborated-type-specifier* is introduced by the *class-key* and this lookup does not find a previously declared *type-name*, or if the *elaborated-type-specifier* appears in a declaration with the form:

class-key identifier attribute-specifier$_{opt}$;

the *elaborated-type-specifier* is a declaration that introduces the *class-name* as described in 3.3.1.

3 If the *elaborated-type-specifier* has a *nested-name-specifier*, qualified name lookup is performed, as described in 3.4.3, but ignoring any non-type names that have been declared. If the name lookup does not find a previously declared *type-name*, the *elaborated-type-specifier* is ill-formed. [*Example:*

```
struct Node {
  struct Node* Next;          // OK: Refers to Node at global scope
  struct Data* Data;          // OK: Declares type Data
                              // at global scope and member Data
};
```

```
struct Data {
  struct Node* Node;             // OK: Refers to Node at global scope
  friend struct ::Glob;          // error: Glob is not declared
                                 // cannot introduce a qualified type (7.1.6.3)
  friend struct Glob;            // OK: Refers to (as yet) undeclared Glob
                // at global scope.
  /* ... */
};

struct Base {
  struct Data;                   // OK: Declares nested Data
  struct ::Data*     thatData;   // OK: Refers to ::Data
  struct Base::Data* thisData;   // OK: Refers to nested Data
  friend class ::Data;           // OK: global Data is a friend
  friend class Data;             // OK: nested Data is a friend
  struct Data { /* ...  */ };    // Defines nested Data
};

struct Data;                     // OK: Redeclares Data at global scope
struct ::Data;                   // error: cannot introduce a qualified type (7.1.6.3)
struct Base::Data;               // error: cannot introduce a qualified type (7.1.6.3)
struct Base::Datum;              // error: Datum undefined
struct Base::Data* pBase;        // OK: refers to nested Data
```

— *end example*]

3.4.5 Class member access [basic.lookup.classref]

1 In a class member access expression (5.2.5), if the `.` or `->` token is immediately followed by an *identifier* followed by a `<`, the identifier must be looked up to determine whether the `<` is the beginning of a template argument list (14.2) or a less-than operator. The identifier is first looked up in the class of the object expression. If the identifier is not found, it is then looked up in the context of the entire *postfix-expression* and shall name a class template. If the lookup in the class of the object expression finds a template, the name is also looked up in the context of the entire *postfix-expression* and

- if the name is not found, the name found in the class of the object expression is used, otherwise
- if the name is found in the context of the entire *postfix-expression* and does not name a class template, the name found in the class of the object expression is used, otherwise
- if the name found is a class template, it shall refer to the same entity as the one found in the class of the object expression, otherwise the program is ill-formed.

2 If the *id-expression* in a class member access (5.2.5) is an *unqualified-id*, and the type of the object expression is of a class type `C`, the *unqualified-id* is looked up in the scope of class `C`. If the type of the object expression is of pointer to scalar type, the *unqualified-id* is looked up in the context of the complete *postfix-expression*.

3 If the *unqualified-id* is ~*type-name*, the *type-name* is looked up in the context of the entire *postfix-expression*. If the type `T` of the object expression is of a class type `C`, the *type-name* is also looked up in the scope of class `C`. At least one of the lookups shall find a name that refers to (possibly cv-qualified) `T`. [*Example:*

```
struct A { };

struct B {
  struct A { };
  void f(::A* a);
};
```

```
void B::f(::A* a) {
  a->~A();                    // OK: lookup in *a finds the injected-class-name
}
```

— *end example*]

4 If the *id-expression* in a class member access is a *qualified-id* of the form

```
class-name-or-namespace-name::...
```

the *class-name-or-namespace-name* following the `.` or `->` operator is looked up both in the context of the entire *postfix-expression* and in the scope of the class of the object expression. If the name is found only in the scope of the class of the object expression, the name shall refer to a *class-name*. If the name is found only in the context of the entire *postfix-expression*, the name shall refer to a *class-name* or *namespace-name*. If the name is found in both contexts, the *class-name-or-namespace-name* shall refer to the same entity.

5 If the *qualified-id* has the form

```
::class-name-or-namespace-name::...
```

the *class-name-or-namespace-name* is looked up in global scope as a *class-name* or *namespace-name*.

6 If the *nested-name-specifier* contains a *simple-template-id* (14.2), the names in its *template-argument*s are looked up in the context in which the entire *postfix-expression* occurs.

7 If the *id-expression* is a *conversion-function-id*, its *conversion-type-id* shall denote the same type in both the context in which the entire *postfix-expression* occurs and in the context of the class of the object expression (or the class pointed to by the pointer expression).

3.4.6 Using-directives and namespace aliases [basic.lookup.udir]

1 When looking up a *namespace-name* in a *using-directive* or *namespace-alias-definition*, only namespace names are considered.

3.5 Program and linkage [basic.link]

1 A *program* consists of one or more *translation units* (Clause 2) linked together. A translation unit consists of a sequence of declarations.

translation-unit:
declaration-seq$_{opt}$

2 A name is said to have *linkage* when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

— When a name has *external linkage*, the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.

— When a name has *internal linkage*, the entity it denotes can be referred to by names from other scopes in the same translation unit.

— When a name has *no linkage*, the entity it denotes cannot be referred to by names from other scopes.

3 A name having namespace scope (3.3.5) has internal linkage if it is the name of

— an object, reference, function or function template that is explicitly declared `static` or,

— an object or reference that is explicitly declared `const` and neither explicitly declared `extern` nor previously declared to have external linkage; or

— a data member of an anonymous union.

4 A name having namespace scope has external linkage if it is the name of

— an object or reference, unless it has internal linkage; or

— a function, unless it has internal linkage; or

— a named class (Clause 9), or an unnamed class defined in a typedef declaration in which the class has the typedef name for linkage purposes (7.1.3); or

— a named enumeration (7.2), or an unnamed enumeration defined in a typedef declaration in which the enumeration has the typedef name for linkage purposes (7.1.3); or

— an enumerator belonging to an enumeration with external linkage; or

— a template, unless it is a function template that has internal linkage (Clause 14); or

— a namespace (7.3), unless it is declared within an unnamed namespace.

5 In addition, a member function, static data member, a named class or enumeration of class scope, or an unnamed class or enumeration defined in a class-scope typedef declaration such that the class or enumeration has the typedef name for linkage purposes (7.1.3), has external linkage if the name of the class has external linkage.

6

The name of a function declared in block scope and the name of an object declared by a block scope `extern` declaration have linkage. If there is a visible declaration of an entity with linkage having the same name and type, ignoring entities declared outside the innermost enclosing namespace scope, the block scope declaration declares that same entity and receives the linkage of the previous declaration. If there is more than one such matching entity, the program is ill-formed. Otherwise, if no matching entity is found, the block scope entity receives external linkage.[*Example:*

```
static void f();
static int i = 0;               // 1
void g() {
  extern void f();              // internal linkage
  int i;                        // 2: i has no linkage
  {
    extern void f();            // internal linkage
    extern int i;               // 3: external linkage
  }
}
```

There are three objects named `i` in this program. The object with internal linkage introduced by the declaration in global scope (line `//1`), the object with automatic storage duration and no linkage introduced by the declaration on line `//2`, and the object with static storage duration and external linkage introduced by the declaration on line `//3`. — *end example*]

7 When a block scope declaration of an entity with linkage is not found to refer to some other declaration, then that entity is a member of the innermost enclosing namespace. However such a declaration does not introduce the member name in its namespace scope. [*Example:*

```
namespace X {
  void p() {
    q();                        // error: q not yet declared
    extern void q();            // q is a member of namespace X
  }
```

```
  void middle() {
    q();                          // error: q not yet declared
  }

  void q() { /* ...  */ }         // definition of X::q
}

void q() { /* ...  */ }           // some other, unrelated q
```

— *end example*]

8 Names not covered by these rules have no linkage. Moreover, except as noted, a name declared in a local scope (3.3.2) has no linkage. A type is said to have linkage if and only if:

— it is a class or enumeration type that is named (or has a name for linkage purposes (7.1.3)) and the name has linkage; or

— it is a specialization of a class template (14)[30]; or

— it is a fundamental type (3.9.1); or

— it is a compound type (3.9.2) other than a class or enumeration, compounded exclusively from types that have linkage; or

— it is a cv-qualified (3.9.3) version of a type that has linkage.

A type without linkage shall not be used as the type of a variable or function with linkage, unless

— the variable or function has extern "C" linkage (7.5), or

— the type without linkage was named using a dependent type (14.6.2.1).

[*Note:* in other words, a type without linkage contains a class or enumeration that cannot be named outside its translation unit. An entity with external linkage declared using such a type could not correspond to any other entity in another translation unit of the program and thus is not permitted. Also note that classes with linkage may contain members whose types do not have linkage, and that typedef names are ignored in the determination of whether a type has linkage. — *end note*] [*Example:*

```
void f() {
  struct A { int x; };            // no linkage
  extern A a;                     // ill-formed
  typedef A B;
  extern B b;                     // ill-formed
}
```

— *end example*]

[*Example:*

```
template <class T> struct A {
  // in A<X>, the following is allowed because the type with no linkage
  // X is named using template parameter T.
  friend void f(A, T){}
};
```

30) A class template always has external linkage, and the requirements of 14.3.1 and 14.3.2 ensure that the template arguments will also have appropriate linkage.

```
template <class T> void g(T t) {
  A<T> at;
  f(at, t);
}

int main() {
  class X {} x;
  g(x);
}
```

— *end example*]

9 Two names that are the same (Clause 3) and that are declared in different scopes shall denote the same object, reference, function, type, enumerator, template or namespace if

— both names have external linkage or else both names have internal linkage and are declared in the same translation unit; and

— both names refer to members of the same namespace or to members, not by inheritance, of the same class; and

— when both names denote functions, the parameter-type-lists of the functions (8.3.5) are identical; and

— when both names denote function templates, the signatures (14.5.6.1) are the same.

10 After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, except that declarations for an array object can specify array types that differ by the presence or absence of a major array bound (8.3.4). A violation of this rule on type identity does not require a diagnostic.

11 [*Note:* linkage to non-C++ declarations can be achieved using a *linkage-specification* (7.5). — *end note*]

3.6 Start and termination [basic.start]

3.6.1 Main function [basic.start.main]

1 A program shall contain a global function called `main`, which is the designated start of the program. It is implementation-defined whether a program in a freestanding environment is required to define a `main` function. [*Note:* in a freestanding environment, start-up and termination is implementation-defined; start-up contains the execution of constructors for objects of namespace scope with static storage duration; termination contains the execution of destructors for objects with static storage duration. — *end note*]

2 An implementation shall not predefine the `main` function. This function shall not be overloaded. It shall have a return type of type `int`, but otherwise its type is implementation-defined. All implementations shall allow both of the following definitions of `main`:

```
int main() { /* ... */ }
```

and

```
int main(int argc, char* argv[]) { /* ... */ }
```

In the latter form `argc` shall be the number of arguments passed to the program from the environment in which the program is run. If `argc` is nonzero these arguments shall be supplied in `argv[0]` through `argv[argc-1]` as pointers to the initial characters of null-terminated multibyte strings (NTMBS s) (17.5.3.2.4.3) and `argv[0]` shall be the pointer to the initial character of a NTMBS that represents the name used to invoke the program or `""`. The value of `argc` shall be nonnegative. The value of `argv[argc]`

shall be 0. [*Note:* it is recommended that any further (optional) parameters be added after `argv`. — *end note*]

3 The function `main` shall not be used (3.2) within a program. The linkage (3.5) of `main` is implementation-defined. A program that declares `main` to be `inline` or `static` is ill-formed. The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. — *end example*]

4 Calling the function `std::exit(int)` declared in `<cstdlib>` (18.4) terminates the program without leaving the current block and hence without destroying any objects with automatic storage duration (12.4). If `std::exit` is called to end a program during the destruction of an object with static or thread storage duration, the program has undefined behavior.

5 A return statement in `main` has the effect of leaving the main function (destroying any objects with automatic storage duration) and calling `std::exit` with the return value as the argument. If control reaches the end of `main` without encountering a `return` statement, the effect is that of executing

```
return 0;
```

3.6.2 Initialization of non-local objects [basic.start.init]

1 There are two broad classes of named non-local objects: those with static storage duration (3.7.1) and those with thread storage duration (3.7.2). Non-local objects with static storage duration are initialized as a consequence of program initiation. Non-local objects with thread storage duration are initialized as a consequence of thread execution. Within each of these phases of initiation, initialization occurs as follows.

2 Objects with static storage duration (3.7.1) or thread storage duration (3.7.2) shall be zero-initialized (8.5) before any other initialization takes place.

Constant initialization is performed:

— if each full-expression (including implicit conversions) that appears in the initializer of a reference with static or thread storage duration is a constant expression (5.19) and the reference is bound to an lvalue designating an object with static storage duration or to a temporary (see 12.2)

— if an object with static or thread storage duration is initialized such that the initialization satisfies the requirements for the object being declared with `constexpr` (7.1.5).

Together, zero-initialization and constant initialization are called *static initialization*; all other initialization is *dynamic initialization*. Static initialization shall be performed before any dynamic initialization takes place. Dynamic initialization of a non-local object with static storage duration is either ordered or unordered. Definitions of explicitly specialized class template static data members have ordered initialization. Other class template static data members (i.e., implicitly or explicitly instantiated specializations) have unordered initialization. Other objects defined in namespace scope have ordered initialization. Objects with ordered initialization defined within a single translation unit shall be initialized in the order of their definitions in the translation unit. If a program starts a thread (30.2), the subsequent initialization of an object is unsequenced with respect to the initialization of an object defined in a different translation unit. Otherwise, the initialization of an object is indeterminately sequenced with respect to the initialization of an object defined in a different translation unit. If a program starts a thread, the subsequent unordered initialization of an object is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of an object is indeterminately sequenced with respect to every other dynamic initialization. [*Note:* This definition permits initialization of a sequence of ordered objects concurrently with another sequence. — *end note*] [*Note:* 8.5.1 describes the order in which aggregate members are initialized. The initialization of local static objects is described in 6.7. — *end note*]

3 An implementation is permitted to perform the initialization of an object of namespace scope as a static initialization even if such initialization is not required to be done statically, provided that

— the dynamic version of the initialization does not change the value of any other object of namespace scope prior to its initialization, and

— the static version of the initialization produces the same value in the initialized object as would be produced by the dynamic initialization if all objects not required to be initialized statically were initialized dynamically.

— [*Note:* as a consequence, if the initialization of an object `obj1` refers to an object `obj2` of namespace scope potentially requiring dynamic initialization and defined later in the same translation unit, it is unspecified whether the value of `obj2` used will be the value of the fully initialized `obj2` (because `obj2` was statically initialized) or will be the value of `obj2` merely zero-initialized. For example,

```
inline double fd() { return 1.0; }
extern double d1;
double d2 = d1;     // unspecified:
                    // may be statically initialized to 0.0 or
                    // dynamically initialized to 1.0
double d1 = fd();   // may be initialized statically to 1.0
```

— *end note*]

4 It is implementation-defined whether the dynamic initialization (8.5, 9.4, 12.1, 12.6.1) of an object of namespace scope with static storage duration is done before the first statement of `main`. If the initialization is deferred to some point in time after the first statement of `main`, it shall occur before the first use of any function or object defined in the same translation unit as the object to be initialized.[31] [*Example:*

```
// - File 1 -
#include "a.h"
#include "b.h"
B b;
A::A(){
  b.Use();
}

// - File 2 -
#include "a.h"
A a;

// - File 3 -
#include "a.h"
#include "b.h"
extern A a;
extern B b;

int main() {
  a.Use();
  b.Use();
}
```

It is implementation-defined whether either `a` or `b` is initialized before `main` is entered or whether the initializations are delayed until `a` is first used in `main`. In particular, if `a` is initialized before `main` is entered,

31) An object defined in namespace scope having initialization with side-effects must be initialized even if it is not used (3.7.1).

it is not guaranteed that `b` will be initialized before it is used by the initialization of `a`, that is, before `A::A` is called. If, however, `a` is initialized at some point after the first statement of `main`, `b` will be initialized prior to its use in `A::A`. — *end example*]

5 It is implementation-defined whether the dynamic initialization (8.5, 9.4, 12.1, 12.6.1) of an object of namespace scope and with thread storage duration is done before the first statement of the initial function of the thread. If the initialization is deferred to some point in time after the first statement of the initial function of the thread, it shall occur before the first use of any object with thread storage duration defined in the same translation unit as the object to be initialized.

6 If construction or destruction of a non-local static or thread duration object ends in throwing an uncaught exception, the result is to call `std::terminate` (18.7.3.3).

3.6.3 Termination [basic.start.term]

1 Destructors (12.4) for initialized objects with static storage duration are called as a result of returning from `main` and as a result of calling `std::exit` (18.4). Destructors for initialized objects with thread storage duration within a given thread are called as a result of returning from the initial function of that thread and as a result of that thread calling `std::exit`. The completions of the destructors for all initialized objects with thread storage duration within that thread are sequenced before the initiation of the destructors of any object with static storage duration. If the completion of the constructor or dynamic initialization of an object with thread storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. If the completion of the constructor or dynamic initialization of an object with static storage duration is sequenced before that of another, the completion of the destructor of the second is sequenced before the initiation of the destructor of the first. [*Note:* this definition permits concurrent destruction. — *end note*] If an object is initialized statically, the object is destroyed in the same order as if the object was dynamically initialized. For an object of array or class type, all subobjects of that object are destroyed before any local object with static storage duration initialized during the construction of the subobjects is destroyed.

2 If a function contains a local object of static or thread storage duration that has been destroyed and the function is called during the destruction of an object with static or thread storage duration, the program has undefined behavior if the flow of control passes through the definition of the previously destroyed local object. Likewise, the behavior is undefined if the function-local object is used indirectly (i.e. through a pointer) after its destruction.

3 If the completion of the initialization of a non-local object with static storage duration is sequenced before a call to `std::atexit` (see `<cstdlib>`, 18.4), the call to the function passed to `std::atexit` is sequenced before the call to the destructor for the object. If a call to `std::atexit` is sequenced before the completion of the initialization of a non-local object with static storage duration, the call to the destructor for the object is sequenced before the call to the function passed to `std::atexit`. If a call to `std::atexit` is sequenced before another call to `std::atexit`, the call to the function passed to the second `std::atexit` call is sequenced before the call to the function passed to the first `std::atexit` call.

4 If there is a use of a standard library object or function not permitted within signal handlers (18.9) that does not happen before (1.10) completion of destruction of objects with static storage duration and execution of `std::atexit` registered functions (18.4), the program has undefined behavior. [*Note:* if there is a use of an object with static storage duration that does not happen before the object's destruction, the program has undefined behavior. Terminating every thread before a call to `std::exit` or the exit from `main` is sufficient, but not necessary, to satisfy these requirements. These requirements permit thread managers as static-storage-duration objects. — *end note*]

5 Calling the function `std::abort()` declared in `<cstdlib>` terminates the program without executing any destructors and without calling the functions passed to `std::atexit()` or `std::at_quick_exit()`.

3.7 Storage duration [basic.stc]

1 Storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object. The storage duration is determined by the construct used to create the object and is one of the following:

- static storage duration
- thread storage duration
- automatic storage duration
- dynamic storage duration

2 Static, thread, and automatic storage durations are associated with objects introduced by declarations (3.1) and implicitly created by the implementation (12.2). The dynamic storage duration is associated with objects created with `operator new` (5.3.4).

3 The storage duration categories apply to references as well. The lifetime of a reference is its storage duration.

3.7.1 Static storage duration [basic.stc.static]

1 All objects which do not have dynamic storage duration, do not have thread storage duration, and are not local have *static storage duration*. The storage for these objects shall last for the duration of the program (3.6.2, 3.6.3).

2 If an object of static storage duration has initialization or a destructor with side effects, it shall not be eliminated even if it appears to be unused, except that a class object or its copy may be eliminated as specified in 12.8.

3 The keyword `static` can be used to declare a local variable with static storage duration. [*Note:* 6.7 describes the initialization of local `static` variables; 3.6.3 describes the destruction of local `static` variables. — *end note*]

4 The keyword `static` applied to a class data member in a class definition gives the data member static storage duration.

3.7.2 Thread storage duration [basic.stc.thread]

1 All objects and references declared with the `thread_local` keyword have *thread storage duration*. The storage for these objects and references shall last for the duration of the thread in which they are created. There is a distinct object or reference per thread, and use of the declared name refers to the object or reference associated with the current thread.

2 An object or reference with thread storage duration shall be initialized before its first use and, if constructed, shall be destroyed on thread exit.

3.7.3 Automatic storage duration [basic.stc.auto]

1 Local objects explicitly declared `register` or not explicitly declared `static` or `extern` have *automatic storage duration*. The storage for these objects lasts until the block in which they are created exits.

2 [*Note:* these objects are initialized and destroyed as described in 6.7. — *end note*]

3 If a named automatic object has initialization or a destructor with side effects, it shall not be destroyed before the end of its block, nor shall it be eliminated as an optimization even if it appears to be unused, except that a class object or its copy may be eliminated as specified in 12.8.

3.7.4 Dynamic storage duration [basic.stc.dynamic]

1 Objects can be created dynamically during program execution (1.9), using *new-expression*s (5.3.4), and destroyed using *delete-expression*s (5.3.5). A C++ implementation provides access to, and management of, dynamic storage via the global *allocation functions* `operator new` and `operator new[]` and the global *deallocation functions* `operator delete` and `operator delete[]`.

2 The library provides default definitions for the global allocation and deallocation functions. Some global allocation and deallocation functions are replaceable (18.5.1). A C++ program shall provide at most one definition of a replaceable allocation or deallocation function. Any such function definition replaces the default version provided in the library (17.6.4.6). The following allocation and deallocation functions (18.5) are implicitly declared in global scope in each translation unit of a program.

```
void* operator new(std::size_t) throw(std::bad_alloc);
void* operator new[](std::size_t) throw(std::bad_alloc);
void operator delete(void*) throw();
void operator delete[](void*) throw();
```

These implicit declarations introduce only the function names `operator new`, `operator new[]`, `operator delete`, `operator delete[]`. [*Note:* the implicit declarations do not introduce the names `std`, `std::bad_alloc`, and `std::size_t`, or any other names that the library uses to declare these names. Thus, a *new-expression*, *delete-expression* or function call that refers to one of these functions without including the header `<new>` is well-formed. However, referring to `std`, `std::bad_alloc`, and `std::size_t` is ill-formed unless the name has been declared by including the appropriate header. — *end note*] Allocation and/or deallocation functions can also be declared and defined for any class (12.5).

3 Any allocation and/or deallocation functions defined in a C++ program, including the default versions in the library, shall conform to the semantics specified in 3.7.4.1 and 3.7.4.2.

3.7.4.1 Allocation functions [basic.stc.dynamic.allocation]

1 An allocation function shall be a class member function or a global function; a program is ill-formed if an allocation function is declared in a namespace scope other than global scope or declared static in global scope. The return type shall be `void*`. The first parameter shall have type `std::size_t` (18.1). The first parameter shall not have an associated default argument (8.3.6). The value of the first parameter shall be interpreted as the requested size of the allocation. An allocation function can be a function template. Such a template shall declare its return type and first parameter as specified above (that is, template parameter types shall not be used in the return type and first parameter type). Template allocation functions shall have two or more parameters.

2 The allocation function attempts to allocate the requested amount of storage. If it is successful, it shall return the address of the start of a block of storage whose length in bytes shall be at least as large as the requested size. There are no constraints on the contents of the allocated storage on return from the allocation function. The order, contiguity, and initial value of storage allocated by successive calls to an allocation function are unspecified. The pointer returned shall be suitably aligned so that it can be converted to a pointer of any complete object type with a fundamental alignment requirement (3.11) and then used to access the object or array in the storage allocated (until the storage is explicitly deallocated by a call to a corresponding deallocation function). Even if the size of the space requested is zero, the request can fail. If the request succeeds, the value returned shall be a non-null pointer value (4.10) `p0` different from any

previously returned value `p1`, unless that value `p1` was subsequently passed to an `operator delete`. The effect of dereferencing a pointer returned as a request for zero size is undefined.[32]

3 An allocation function that fails to allocate storage can invoke the currently installed new-handler function (18.5.2.2), if any. [*Note:* A program-supplied allocation function can obtain the address of the currently installed `new_handler` using the `std::set_new_handler` function (18.5.2.3). — *end note*] If an allocation function declared with an empty *exception-specification* (15.4), `throw()`, fails to allocate storage, it shall return a null pointer. Any other allocation function that fails to allocate storage shall indicate failure only by throwing an exception of a type that would match a handler (15.3) of type `std::bad_alloc` (18.5.2.1).

4 A global allocation function is only called as the result of a new expression (5.3.4), or called directly using the function call syntax (5.2.2), or called indirectly through calls to the functions in the C++ standard library. [*Note:* in particular, a global allocation function is not called to allocate storage for objects with static storage duration (3.7.1), for objects or references with thread storage duration (3.7.2) for objects of type `std::type_info` (5.2.8), or for the copy of an object thrown by a `throw` expression (15.1). — *end note*]

3.7.4.2 Deallocation functions [basic.stc.dynamic.deallocation]

1 Deallocation functions shall be class member functions or global functions; a program is ill-formed if deallocation functions are declared in a namespace scope other than global scope or declared static in global scope.

2 Each deallocation function shall return `void` and its first parameter shall be `void*`. A deallocation function can have more than one parameter. If a class `T` has a member deallocation function named `operator delete` with exactly one parameter, then that function is a usual (non-placement) deallocation function. If class `T` does not declare such an `operator delete` but does declare a member deallocation function named `operator delete` with exactly two parameters, the second of which has type `std::size_t` (18.1), then this function is a usual deallocation function. Similarly, if a class `T` has a member deallocation function named `operator delete[]` with exactly one parameter, then that function is a usual (non-placement) deallocation function. If class `T` does not declare such an `operator delete[]` but does declare a member deallocation function named `operator delete[]` with exactly two parameters, the second of which has type `std::size_t`, then this function is a usual deallocation function. A deallocation function can be an instance of a function template. Neither the first parameter nor the return type shall depend on a template parameter. [*Note:* that is, a deallocation function template shall have a first parameter of type `void*` and a return type of `void` (as specified above). — *end note*] A deallocation function template shall have two or more function parameters. A template instance is never a usual deallocation function, regardless of its signature.

3 If a deallocation function terminates by throwing an exception, the behavior is undefined. The value of the first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation function is one supplied in the standard library, the call has no effect. Otherwise, the value supplied to `operator delete(void*)` in the standard library shall be one of the values returned by a previous invocation of either `operator new(std::size_t)` or `operator new(std::size_t, const std::nothrow_t&)` in the standard library, and the value supplied to `operator delete[](void*)` in the standard library shall be one of the values returned by a previous invocation of either `operator new[](std::size_t)` or `operator new[](std::size_t, const std::nothrow_t&)` in the standard library.

4 If the argument given to a deallocation function in the standard library is a pointer that is not the null pointer value (4.10), the deallocation function shall deallocate the storage referenced by the pointer, rendering invalid

32) The intent is to have `operator new()` implementable by calling `std::malloc()` or `std::calloc()`, so the rules are substantially the same. C++ differs from C in requiring a zero request to return a non-null pointer.

all pointers referring to any part of the *deallocated storage*. The effect of using an invalid pointer value (including passing it to a deallocation function) is undefined.[33]

3.7.4.3 Safely-derived pointers [basic.stc.dynamic.safety]

1 A *traceable pointer object* is

- an object of pointer-to-object type, or
- an object of an integral type that is at least as large as `std::intptr_t`, or
- a sequence of elements in an array of character type, where the size and alignment of the sequence match that of some pointer-to-object type.

2 A pointer value is a *safely-derived pointer* to a dynamic object only if it has pointer-to-object type and it is one of the following:

- the value returned by a call to the C++ standard library implementation of `::operator new(std::size_t)`;[34]
- the result of taking the address of a subobject of an lvalue resulting from dereferencing a safely-derived pointer value;
- the result of well-defined pointer arithmetic using a safely-derived pointer value;
- the result of a well-defined pointer conversion of a safely-derived pointer value;
- the result of a `reinterpret_cast` of a safely-derived pointer value;
- the result of a `reinterpret_cast` of an integer representation of a safely-derived pointer value;
- the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained a copy of a safely-derived pointer value.

3 An integer value is an *integer representation of a safely-derived pointer* only if its type is at least as large as `std::intptr_t` and it is one of the following:

- the result of a `reinterpret_cast` of a safely-derived pointer value;
- the result of a valid conversion of an integer representation of a safely-derived pointer value;
- the value of an object whose value was copied from a traceable pointer object, where at the time of the copy the source object contained an integer representation of a safely-derived pointer value;
- the result of an additive or bitwise operation, one of whose operands is an integer representation of a safely-derived pointer value `P`, if that result converted by `reinterpret_cast<void*>` would compare equal to a safely-derived pointer computable from `reinterpret_cast<void*>(P)`.

4 If a pointer value that is not a safely-derived pointer value is dereferenced or deallocated, and the referenced complete object is of dynamic storage duration and has not previously been declared reachable (20.7.13.7), the behavior is undefined. [*Note:* this is true even if the unsafely-derived pointer value might compare equal to some safely-derived pointer value. — *end note*]

33) On some implementations, it causes a system-generated runtime fault.

34) This section does not impose restrictions on dereferencing pointers to memory not allocated by `::operator new`. This maintains the ability of many C++ implementations to use binary libraries and components written in other languages. In particular, this applies to C binaries, because dereferencing pointers to memory allocated by `malloc` is not restricted.

3.7.5 Duration of subobjects [basic.stc.inherit]

1 The storage duration of member subobjects, base class subobjects and array elements is that of their complete object (1.8).

3.8 Object lifetime [basic.life]

1 The *lifetime* of an object is a runtime property of the object. An object is said to have non-trivial initialization if it is of a class or aggregate type and it or one of its members is initialized by a constructor other than a trivial default constructor. [*Note:* initialization by a trivial copy constructor is non-trivial initialization. —*end note*] The lifetime of an object of type `T` begins when:

— storage with the proper alignment and size for type `T` is obtained, and

— if the object has non-trivial initialization, its initialization is complete.

The lifetime of an object of type `T` ends when:

— if `T` is a class type with a non-trivial destructor (12.4), the destructor call starts, or

— the storage which the object occupies is reused or released.

2 [*Note:* the lifetime of an array object starts as soon as storage with proper size and alignment is obtained, and its lifetime ends when the storage which the array occupies is reused or released. 12.6.2 describes the lifetime of base and member subobjects. —*end note*]

3 The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. [*Note:* in particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 12.6.2 and in 12.7. Also, the behavior of an object under construction and destruction might not be the same as the behavior of an object whose lifetime has started and not ended. 12.6.2 and 12.7 describe the behavior of objects during the construction and destruction phases. —*end note*]

4 A program may end the lifetime of any object by reusing the storage which the object occupies or by explicitly calling the destructor for an object of a class type with a non-trivial destructor. For an object of a class type with a non-trivial destructor, the program is not required to call the destructor explicitly before the storage which the object occupies is reused or released; however, if there is no explicit call to the destructor or if a *delete-expression* (5.3.5) is not used to release the storage, the destructor shall not be implicitly called and any program that depends on the side effects produced by the destructor has undefined behavior.

5 Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[35] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. Such a pointer refers to allocated storage (3.7.4.2), and using the pointer as if the pointer were of type `void*`, is well-defined. Such a pointer may be dereferenced but the resulting lvalue may only be used in limited ways, as described below. The program has undefined behavior if:

— the object will be or was of a class type with a non-trivial destructor and the pointer is used as the operand of a *delete-expression*,

— the pointer is used to access a non-static data member or call a non-static member function of the object, or

— the pointer is implicitly converted (4.10) to a pointer to a base class type, or

35) For example, before the construction of a global object of non-POD class type (12.7).

— the pointer is used as the operand of a `static_cast` (5.2.9) (except when the conversion is to `void*`, or to `void*` and subsequently to `char*`, or `unsigned char*`), or

— the pointer is used as the operand of a `dynamic_cast` (5.2.7). [*Example:*

```
#include <cstdlib>

struct B {
  virtual void f();
  void mutate();
  virtual ~B();
};

struct D1 : B { void f(); };
struct D2 : B { void f(); };

void B::mutate() {
  new (this) D2;     // reuses storage — ends the lifetime of *this
  f();               // undefined behavior
  ... = this;        // OK, this points to valid memory
}

void g() {
  void* p = std::malloc(sizeof(D1) + sizeof(D2));
  B* pb = new (p) D1;
  pb->mutate();
  &pb;               // OK: pb points to valid memory
  void* q = pb;      // OK: pb points to valid memory
  pb->f();           // undefined behavior, lifetime of *pb has ended
}
```

— *end example*]

6 Similarly, before the lifetime of an object has started but after the storage which the object will occupy has been allocated or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any lvalue which refers to the original object may be used but only in limited ways. Such an lvalue refers to allocated storage (3.7.4.2), and using the properties of the lvalue which do not depend on its value is well-defined. The program has undefined behavior if:

— an lvalue-to-rvalue conversion (4.1) is applied to such an lvalue,

— the lvalue is used to access a non-static data member or call a non-static member function of the object, or

— the lvalue is implicitly converted (4.10) to a reference to a base class type, or

— the lvalue is used as the operand of a `static_cast` (5.2.9) except when the conversion is ultimately to *cv* `char&` or *cv* `unsigned char&`, or

— the lvalue is used as the operand of a `dynamic_cast` (5.2.7) or as the operand of `typeid`.

7 If, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, a new object is created at the storage location which the original object occupied, a pointer that pointed to the original object, a reference that referred to the original object, or the name of the original object will automatically refer to the new object and, once the lifetime of the new object has started, can be used to manipulate the new object, if:

— the storage for the new object exactly overlays the storage location which the original object occupied, and

— the new object is of the same type as the original object (ignoring the top-level cv-qualifiers), and

— the type of the original object is not const-qualified, and, if a class type, does not contain any non-static data member whose type is const-qualified or a reference type, and

— the original object was a most derived object (1.8) of type `T` and the new object is a most derived object of type `T` (that is, they are not base class subobjects). [*Example:*

```
struct C {
  int i;
  void f();
  const C& operator=( const C& );
};

const C& C::operator=( const C& other) {
  if ( this != &other ) {
    this->~C();                 // lifetime of *this ends
    new (this) C(other);        // new object of type C created
    f();                        // well-defined
  }
  return *this;
}

C c1;
C c2;
c1 = c2;                        // well-defined
c1.f();                         // well-defined; c1 refers to a new object of type C
```

— *end example*]

8 If a program ends the lifetime of an object of type `T` with static (3.7.1), thread (3.7.2), or automatic (3.7.3) storage duration and if `T` has a non-trivial destructor,[36] the program must ensure that an object of the original type occupies that same storage location when the implicit destructor call takes place; otherwise the behavior of the program is undefined. This is true even if the block is exited with an exception. [*Example:*

```
class T { };
struct B {
   ~B();
};

void h() {
   B b;
   new (&b) T;
}                               // undefined behavior at block exit
```

— *end example*]

9 Creating a new object at the storage location that a `const` object with static, thread, or automatic storage duration occupies or, at the storage location that such a `const` object used to occupy before its lifetime ended results in undefined behavior. [*Example:*

36) That is, an object for which a destructor will be called implicitly—upon exit from the block for an object with automatic storage duration, upon exit from the thread for an object with thread storage duration, or upon exit from the program for an object with static storage duration.

```
struct B {
  B();
  ~B();
};

const B b;

void h() {
  b.~B();
  new (&b) const B;              // undefined behavior
}
```

— *end example*]

3.9 Types [basic.types]

1 [*Note:* 3.9 and the subclauses thereof impose requirements on implementations regarding the representation of types. There are two kinds of types: fundamental types and compound types. Types describe objects (1.8), references (8.3.2), or functions (8.3.5). In a constrained context (14.10), type archetypes are class types; however, a type archetype that has a compiler-supported requirement (i.e., a type archetype T' in a context where some requirement C<T> is in scope) may be used in certain contexts where a class type is normallly not permitted or may be disallowed in certain contexts where a class type is normally permitted. Such contexts are explicitly noted in this International Standard. — *end note*]

2 For any object (other than a base-class subobject) of trivially copyable type `T`, whether the object holds a valid value of type `T`, the underlying bytes (1.7) making up the object can be copied into an array of `char` or `unsigned char`.[37] If the content of the array of `char` or `unsigned char` is copied back into the object, the object shall subsequently hold its original value. [*Example:*

```
#define N sizeof(T)
char buf[N];
T obj;                           // obj initialized to its original value
std::memcpy(buf, &obj, N);       // between these two calls to std::memcpy,
                                 // obj might be modified
std::memcpy(&obj, buf, N);       // at this point, each subobject of obj of scalar type
                                 // holds its original value
```

— *end example*]

3 For any trivially copyable type `T`, if two pointers to `T` point to distinct `T` objects `obj1` and `obj2`, where neither `obj1` nor `obj2` is a base-class subobject, if the value of `obj1` is copied into `obj2`, using the `std::memcpy` library function, `obj2` shall subsequently hold the same value as `obj1`. [*Example:*

```
T* t1p;
T* t2p;
    // provided that t2p points to an initialized object ...
std::memcpy(t1p, t2p, sizeof(T));
    // at this point, every subobject of trivially copyable type in *t1p contains
    // the same value as the corresponding subobject in *t2p
```

— *end example*]

4 The *object representation* of an object of type `T` is the sequence of N `unsigned char` objects taken up by the object of type `T`, where N equals `sizeof(T)`. The *value representation* of an object is the set of bits that

37) By using, for example, the library functions (17.6.2.3) `std::memcpy` or `std::memmove`.

hold the value of type T. For trivially copyable types, the value representation is a set of bits in the object representation that determines a *value*, which is one discrete element of an implementation-defined set of values.[38]

5 A class that has been declared but not defined, or an array of unknown size or of incomplete element type, is an incompletely-defined object type.[39] Incompletely-defined object types and the void types are incomplete types (3.9.1). Objects shall not be defined to have an incomplete type.

6 A class type (such as "class X") might be incomplete at one point in a translation unit and complete later on; the type "class X" is the same type at both points. The declared type of an array object might be an array of incomplete class type and therefore incomplete; if the class type is completed later on in the translation unit, the array type becomes complete; the array type at those two points is the same type. The declared type of an array object might be an array of unknown size and therefore be incomplete at one point in a translation unit and complete later on; the array types at those two points ("array of unknown bound of T" and "array of N T") are different types. The type of a pointer to array of unknown size, or of a type defined by a typedef declaration to be an array of unknown size, cannot be completed. [*Example:*

```
class X;                        // X is an incomplete type
extern X* xp;                   // xp is a pointer to an incomplete type
extern int arr[];               // the type of arr is incomplete
typedef int UNKA[];             // UNKA is an incomplete type
UNKA* arrp;                     // arrp is a pointer to an incomplete type
UNKA** arrpp;

void foo() {
  xp++;                         // ill-formed: X is incomplete
  arrp++;                       // ill-formed: incomplete type
  arrpp++;                      // OK: sizeof UNKA* is known
}

struct X { int i; };            // now X is a complete type
int  arr[10];                   // now the type of arr is complete

X x;
void bar() {
  xp = &x;                      // OK; type is "pointer to X"
  arrp = &arr;                  // ill-formed: different types
  xp++;                         // OK: X is complete
  arrp++;                       // ill-formed: UNKA can't be completed
}
```

— *end example*]

7 [*Note:* the rules for declarations and expressions describe in which contexts incomplete types are prohibited. — *end note*]

8 An *object type* is a (possibly cv-qualified) type that is not a function type, not a reference type, and not a void type.

9 An *effective object type* T is a non-archetype object type or a (possibly cv-qualified) type archetype (14.10.2) that has the requirement std::ObjectType<T>.

10 Arithmetic types (3.9.1), enumeration types, pointer types, pointer to member types (3.9.2), and std::nullptr_t, and cv-qualified versions of these types (3.9.3) are collectively called *scalar types*. Scalar types, POD

38) The intent is that the memory model of C++ is compatible with that of ISO/IEC 9899 Programming Language C.
39) The size and layout of an instance of an incompletely-defined object type is unknown.

classes (Clause 9), arrays of such types and *cv-qualified* versions of these types (3.9.3) are collectively called *POD types*. Scalar types, trivially copyable class types (Clause 9), arrays of such types, and cv-qualified versions of these types (3.9.3) are collectively called *trivially copyable types*. Scalar types, trivial class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called *trivial types*. Scalar types, standard-layout class types (Clause 9), arrays of such types and cv-qualified versions of these types (3.9.3) are collectively called *standard-layout types*.

11 An *effective trivial type* `T` is a trivial type or a (possibly cv-qualified) type archetype (14.10.2) that has the requirement `std::TrivialType<T>`.

12 A type is a *literal type* if it is:

- a scalar type; or
- a class type (Clause 9) with
  - a trivial copy constructor,
  - a trivial destructor,
  - a trivial default constructor or at least one constexpr constructor other than the copy constructor, and
  - all non-static data members and base classes of literal types; or
- an array of literal type.

13 An *effective literal type* `T` is a non-archetype literal type or a (possibly cv-qualified) type archetype (14.10.2) that has the requirement `std::LiteralType<T>`.

14 If two types `T1` and `T2` are the same type, then `T1` and `T2` are *layout-compatible* types. [*Note:* Layout-compatible enumerations are described in 7.2. Layout-compatible standard-layout structs and standard-layout unions are described in 9.2. — *end note*]

3.9.1 Fundamental types [basic.fundamental]

1 Objects declared as characters (`char`) shall be large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a `char` object can hold negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements (3.11); that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain `char` object can take on either the same values as a `signed char` or an `unsigned char`; which one is implementation-defined.

2 There are five *standard signed integer types* : "`signed char`", "`short int`", "`int`", "`long int`", and "`long long int`". In this list, each type provides at least as much storage as those preceding it in the list. There may also be implementation-defined *extended signed integer types*. The standard and extended signed integer types are collectively called *signed integer types*. Plain `int`s have the natural size suggested by the architecture of the execution environment[40]; the other signed integer types are provided to meet special needs.

40) that is, large enough to contain any value in the range of `INT_MIN` and `INT_MAX`, as defined in the header `<climits>`.

3 For each of the standard signed integer types, there exists a corresponding (but different) *standard unsigned integer type*: "`unsigned char`", "`unsigned short int`", "`unsigned int`", "`unsigned long int`", and "`unsigned long long int`", each of which occupies the same amount of storage and has the same alignment requirements (3.11) as the corresponding signed integer type[41]; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. Likewise, for each of the extended signed integer types there exists a corresponding *extended unsigned integer type* with the same amount of storage and alignment requirements. The standard and extended unsigned integer types are collectively called *unsigned integer types*. The range of nonnegative values of a *signed integer* type is a subrange of the corresponding *unsigned integer* type, and the value representation of each corresponding signed/unsigned type shall be the same. The standard signed integer types and standard unsigned integer types are collectively called the *standard integer types*, and the extended signed integer types and extended unsigned integer types are collectively called the *extended integer types*.

4 Unsigned integers, declared `unsigned`, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.[42]

5 Type `wchar_t` is a distinct type whose values can represent distinct codes for all members of the largest extended character set specified among the supported locales (22.1.1). Type `wchar_t` shall have the same size, signedness, and alignment requirements (3.11) as one of the other integral types, called its *underlying type*. Types `char16_t` and `char32_t` denote distinct types with the same size, signedness, and alignment as `uint_least16_t` and `uint_least32_t`, respectively, in `<stdint.h>`, called the underlying types.

6 Values of type `bool` are either `true` or `false`.[43] [*Note:* there are no `signed`, `unsigned`, `short`, or `long bool` types or values. — *end note*] Values of type `bool` participate in integral promotions (4.5).

7 Types `bool`, `char`, `char16_t`, `char32_t`, `wchar_t`, and the signed and unsigned integer types are collectively called *integral* types.[44] A synonym for integral type is *integer type*. The representations of integral types shall define values by use of a pure binary numeration system.[45] [*Example:* this International Standard permits 2's complement, 1's complement and signed magnitude representations for integral types. — *end example*]

8 An *effective integral type* `T` is an integral type or a (possibly cv-qualified) type archetype (14.10.2) that has the requirement `std::IntegralType<T>`.

9 There are three *floating point* types: `float`, `double`, and `long double`. The type `double` provides at least as much precision as `float`, and the type `long double` provides at least as much precision as `double`. The set of values of the type `float` is a subset of the set of values of the type `double`; the set of values of the type `double` is a subset of the set of values of the type `long double`. The value representation of floating-point types is implementation-defined. *Integral* and *floating* types are collectively called *arithmetic* types. Specializations of the standard template `std::numeric_limits` (18.2) shall specify the maximum and minimum values of each arithmetic type for an implementation.

10 The `void` type has an empty set of values. The `void` type is an incomplete type that cannot be completed. It is used as the return type for functions that do not return a value. Any expression can be explicitly converted to type *cv* `void` (5.4). An expression of type `void` shall be used only as an expression statement (6.2), as an

41) See 7.1.6.2 regarding the correspondence between types and the sequences of *type-specifier*s that designate them.

42) This implies that unsigned arithmetic does not overflow because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting unsigned integer type.

43) Using a `bool` value in ways described by this International Standard as "undefined," such as by examining the value of an uninitialized automatic variable, might cause it to behave as if it is neither `true` nor `false`.

44) Therefore, enumerations (7.2) are not integral; however, enumerations can be promoted to integral types as specified in 4.5.

45) A positional representation for integers that uses the binary digits 0 and 1, in which the values represented by successive bits are additive, begin with 1, and are multiplied by successive integral power of 2, except perhaps for the bit with the highest position. (Adapted from the *American National Dictionary for Information Processing Systems*.)

operand of a comma expression (5.18), as a second or third operand of `?:` (5.16), as the operand of `typeid`, or as the expression in a return statement (6.6.3) for a function with the return type `void`.

11 A value of type `std::nullptr_t` is a null pointer constant (4.10). Such values participate in the pointer and the pointer to member conversions (4.10, 4.11). `sizeof(std::nullptr_t)` shall be equal to `sizeof(void*)`.

12 [*Note:* even if the implementation defines two or more basic types to have the same value representation, they are nevertheless different types. — *end note*]

3.9.2 Compound types [basic.compound]

1 Compound types can be constructed in the following ways:

- *arrays* of objects of a given type, 8.3.4;
- *functions*, which have parameters of given types and return `void` or references or objects of a given type, 8.3.5;
- *pointers* to `void` or objects or functions (including static members of classes) of a given type, 8.3.1;
- *references* to objects or functions of a given type, 8.3.2. There are two types of references:
  - *lvalue reference*
  - *rvalue reference*
- *classes* containing a sequence of objects of various types (Clause 9), a set of types, enumerations and functions for manipulating these objects (9.3), and a set of restrictions on the access to these entities (Clause 11);
- *unions*, which are classes capable of containing objects of different types at different times, 9.5;
- *enumerations*, which comprise a set of named constant values. Each distinct enumeration constitutes a different *enumerated type*, 7.2;
- *pointers to non-static* [46] *class members*, which identify members of a given type within objects of a given class, 8.3.3.

2 These methods of constructing types can be applied recursively; restrictions are mentioned in 8.3.1, 8.3.4, 8.3.5, and 8.3.2.

3 A pointer to objects of type `T` is referred to as a "pointer to `T`." [*Example:* a pointer to an object of type `int` is referred to as "pointer to `int` " and a pointer to an object of class `X` is called a "pointer to `X`." — *end example*] Except for pointers to static members, text referring to "pointers" does not apply to pointers to members. Pointers to incomplete types are allowed although there are restrictions on what can be done with them (3.11). A valid value of an object pointer type represents either the address of a byte in memory (1.7) or a null pointer (4.10). If an object of type `T` is located at an address `A`, a pointer of type *cv* `T*` whose value is the address `A` is said to *point to* that object, regardless of how the value was obtained. [*Note:* for instance, the address one past the end of an array (5.7) would be considered to point to an unrelated object of the array's element type that might be located at that address. There are further restrictions on pointers to objects with dynamic storage duration; see 3.7.4.3. — *end note*] The value representation of pointer types is implementation-defined. Pointers to cv-qualified and cv-unqualified versions (3.9.3) of layout-compatible types shall have the same value representation and alignment requirements (3.11). [*Note:* pointers to over-aligned types have no special representation, but their range of valid values is restricted by the extended alignment requirement. This International Standard specifies only two ways of obtaining such a pointer: taking the address of a valid object with an over-aligned type, and using one of the runtime

46) Static class members are objects or functions, and pointers to them are ordinary pointers to objects or functions.

pointer alignment functions. An implementation may provide other means of obtaining a valid pointer value for an over-aligned type. — *end note*]

4 Objects of cv-qualified (3.9.3) or cv-unqualified type `void*` (pointer to void), can be used to point to objects of unknown type. A `void*` shall be able to hold any object pointer. A cv-qualified or cv-unqualified (3.9.3) `void*` shall have the same representation and alignment requirements as a cv-qualified or cv-unqualified `char*`.

3.9.3 CV-qualifiers [basic.type.qualifier]

1 A type mentioned in 3.9.1 and 3.9.2 is a *cv-unqualified type*. Each type which is a cv-unqualified complete or incomplete object type or is `void` (3.9) has three corresponding cv-qualified versions of its type: a *const-qualified* version, a *volatile-qualified* version, and a *const-volatile-qualified* version. The term *object type* (1.8) includes the cv-qualifiers specified when the object is created. The presence of a `const` specifier in a *decl-specifier-seq* declares an object of *const-qualified object type*; such object is called a *const object*. The presence of a `volatile` specifier in a *decl-specifier-seq* declares an object of *volatile-qualified object type*; such object is called a *volatile object*. The presence of both *cv-qualifiers* in a *decl-specifier-seq* declares an object of *const-volatile-qualified object type*; such object is called a *const volatile object*. The cv-qualified or cv-unqualified versions of a type are distinct types; however, they shall have the same representation and alignment requirements (3.9).[47]

2 A compound type (3.9.2) is not cv-qualified by the cv-qualifiers (if any) of the types from which it is compounded. Any cv-qualifiers applied to an array type affect the array element type, not the array type (8.3.4).

3 Each non-static, non-mutable, non-reference data member of a const-qualified class object is const-qualified, each non-static, non-reference data member of a volatile-qualified class object is volatile-qualified and similarly for members of a const-volatile class. See 8.3.5 and 9.3.2 regarding cv-qualified function types.

4 There is a (partial) ordering on cv-qualifiers, so that a type can be said to be *more cv-qualified* than another. Table 8 shows the relations that constitute this ordering.

Table 8 — Relations on `const` and `volatile`

no cv-qualifier	<	`const`
no cv-qualifier	<	`volatile`
no cv-qualifier	<	`const volatile`
`const`	<	`const volatile`
`volatile`	<	`const volatile`

5 In this International Standard, the notation *cv* (or *cv1*, *cv2*, etc.), used in the description of types, represents an arbitrary set of cv-qualifiers, i.e., one of {`const` }, {`volatile` }, {`const`, `volatile`}, or the empty set. Cv-qualifiers applied to an array type attach to the underlying element type, so the notation "*cv* `T`," where `T` is an array type, refers to an array whose elements are so-qualified. Such array types can be said to be more (or less) cv-qualified than other types based on the cv-qualification of the underlying element types.

3.10 Lvalues and rvalues [basic.lval]

1 Every expression is either an *lvalue* or an *rvalue*.

47) The same representation and alignment requirements are meant to imply interchangeability as arguments to functions, return values from functions, and members of unions.

2 An lvalue refers to an object or function. Some rvalue expressions—those of (possibly cv-qualified) class or array type—also refer to objects.[48]

3 [*Note:* some built-in operators and function calls yield lvalues. [*Example:* if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object or function to which `E` points. As another example, the function

```
int& f();
```

yields an lvalue, so the call `f()` is an lvalue expression. —*end example*] —*end note*]

4 [*Note:* some built-in operators expect lvalue operands. [*Example:* built-in assignment operators all expect their left-hand operands to be lvalues. —*end example*] Other built-in operators yield rvalues, and some expect them. [*Example:* the unary and binary `+` operators expect rvalue arguments and yield rvalue results. —*end example*] The discussion of each built-in operator in Clause 5 indicates whether it expects lvalue operands and whether it yields an lvalue. —*end note*]

5 The result of calling a function that does not return an lvalue reference is an rvalue. User defined operators are functions, and whether such operators expect or yield lvalues is determined by their parameter and return types.

6 An expression which holds a temporary object resulting from a cast to a type other than an lvalue reference type is an rvalue (this includes the explicit creation of an object using functional notation (5.2.3)).

7 Whenever an lvalue appears in a context where an rvalue is expected, the lvalue is converted to an rvalue; see 4.1, 4.2, and 4.3.

8 The discussion of reference initialization in 8.5.3 and of temporaries in 12.2 indicates the behavior of lvalues and rvalues in other significant contexts.

9 Class rvalues can have cv-qualified types; non-class rvalues always have cv-unqualified types. Rvalues shall always have complete types or the `void` type; in addition to these types, lvalues can also have incomplete types.

10 An lvalue for an object is necessary in order to modify the object except that an rvalue of class type can also be used to modify its referent under certain circumstances. [*Example:* a member function called for an object (9.3) can modify the object. —*end example*]

11 Functions cannot be modified, but pointers to functions can be modifiable.

12 A pointer to an incomplete type can be modifiable. At some point in the program when the pointed to type is complete, the object at which the pointer points can also be modified.

13 The referent of a `const`-qualified expression shall not be modified (through that expression), except that if it is of class type and has a `mutable` component, that component can be modified (7.1.6.1).

14 If an expression can be used to modify the object to which it refers, the expression is called *modifiable*. A program that attempts to modify an object through a nonmodifiable lvalue or rvalue expression is ill-formed.

15 If a program attempts to access the stored value of an object through an lvalue of other than one of the following types the behavior is undefined[49]

— the dynamic type of the object,

— a cv-qualified version of the dynamic type of the object,

48) Expressions such as invocations of constructors and of functions that return a class type refer to objects, and the implementation can invoke a member function upon such objects, but the expressions are not lvalues.

49) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

— a type similar (as defined in 4.4) to the dynamic type of the object,

— a type that is the signed or unsigned type corresponding to the dynamic type of the object,

— a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union),

— a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,

— a `char` or `unsigned char` type.

3.11 Alignment [basic.align]

1 Object types have *alignment requirements* (3.9.1, 3.9.2) which place restrictions on the addressses at which an object of that type may be allocated. An *aligment* is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated. An object type imposes an alignment requirement on every object of that type; stricter alignment can be requested using the alignment attribute (7.6.2).

2 A *fundamental alignment* is represented by an alignment less than or equal to the greatest alignment supported by the implementation in all contexts, which is equal to `alignof(std::max_align_t)` (18.1).

3 An *extended alignment* is represented by an alignment greater than `alignof(std::max_align_t)`. It is implementation-defined whether any extended alignments are supported and the contexts in which they are supported (7.6.2). A type having an extended alignment requirement is an *over-aligned type*. [*Note:* every over-aligned type is or contains a class type with a non-static data member to which an extended alignment has been applied. — *end note*]

4 Alignments are represented as values of the type `std::size_t`. Valid alignments include only those values returned by an `alignof` expression for the fundamental types plus an additional implementation-defined set of values which may be empty.[50]

5 Alignments have an order from *weaker* to *stronger* or *stricter* alignments. Stricter alignments have larger alignment values. An address that satisfies an alignment requirement also satisfies any weaker valid alignment requirement.

6 The alignment requirement of a complete type can be queried using an `alignof` expression (5.3.6). Furthermore, the types `char`, `signed char`, and `unsigned char` shall have the weakest alignment requirement. [*Note:* this enables the character types to be used as the underlying type for an aligned memory area (7.6.2). — *end note*]

7 Comparing alignments is meaningful and provides the obvious results:

— Two alignments are equal when their numeric values are equal.

— Two alignments are different when their numeric values are not equal.

— When an alignment is larger than another it represents a stricter alignment.

8 [*Note:* the runtime pointer alignment function (20.7.14) can be used to obtain an aligned pointer within a buffer; the aligned-storage templates in the library (20.5.7) can be used to obtain aligned storage. — *end note*]

50) It is intended that every valid alignment value be an integral power of two.

9 If a request for a specific extended alignment in a specific context is not supported by an implementation, the program is ill-formed. Additionally, a request for runtime allocation of dynamic storage for which the requested alignment cannot be honored shall be treated as an allocation failure.

4 Standard conversions [conv]

1 Standard conversions are implicit conversions defined for built-in types. Clause 4 enumerates the full set of such conversions. A *standard conversion sequence* is a sequence of standard conversions in the following order:

— Zero or one conversion from the following set: lvalue-to-rvalue conversion, array-to-pointer conversion, and function-to-pointer conversion.

— Zero or one conversion from the following set: integral promotions, floating point promotion, integral conversions, floating point conversions, floating-integral conversions, pointer conversions, pointer to member conversions, and boolean conversions.

— Zero or one qualification conversion.

[*Note:* a standard conversion sequence can be empty, i.e., it can consist of no conversions. — *end note*] A standard conversion sequence will be applied to an expression if necessary to convert it to a required destination type.

2 [*Note:* expressions with a given type will be implicitly converted to other types in several contexts:

— When used as operands of operators. The operator's requirements for its operands dictate the destination type (Clause 5).

— When used in the condition of an `if` statement or iteration statement (6.4, 6.5). The destination type is `bool`.

— When used in the expression of a `switch` statement. The destination type is integral (6.4).

— When used as the source expression for an initialization (which includes use as an argument in a function call and use as the expression in a `return` statement). The type of the entity being initialized is (generally) the destination type. See 8.5, 8.5.3.

— *end note*]

3 An expression `e` can be *implicitly converted* to a type `T` if and only if the declaration `T t=e;` is well-formed, for some invented temporary variable `t` (8.5). Certain language constructs require that an expression be converted to a Boolean value. An expression `e` appearing in such a context is said to be *contextually converted to* `bool` and is well-formed if and only if the declaration `bool t(e);` is well-formed, for some invented temporary variable `t` (8.5). The effect of either implicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if `T` is an lvalue reference type (8.3.2), and an rvalue otherwise. The expression `e` is used as an lvalue if and only if the initialization uses it as an lvalue.

4 [*Note:* For user-defined types, user-defined conversions are considered as well; see 12.3. In general, an implicit conversion sequence (13.3.3.1) consists of a standard conversion sequence followed by a user-defined conversion followed by another standard conversion sequence. — *end note*]

5 [*Note:* There are some contexts where certain conversions are suppressed. For example, the lvalue-to-rvalue conversion is not done on the operand of the unary `&` operator. Specific exceptions are given in the descriptions of those operators and contexts. — *end note*]

4.1 Lvalue-to-rvalue conversion [conv.lval]

1 An lvalue (3.10) of a non-function, non-array type `T` can be converted to an rvalue. If `T` is an incomplete type, a program that necessitates this conversion is ill-formed. If the object to which the lvalue refers is not an object of type `T` and is not an object of a type derived from `T`, or if the object is uninitialized, a program that necessitates this conversion has undefined behavior. If `T` is not an effective class type, the type of the rvalue is the cv-unqualified version of `T`. Otherwise, the type of the rvalue is `T`.[51]

2 When an lvalue-to-rvalue conversion occurs in an unevaluated operand or a subexpression thereof (Clause 5) the value contained in the referenced object is not accessed. Otherwise, if the lvalue has a class type, the conversion copy-initializes a temporary of type `T` from the lvalue and the result of the conversion is an rvalue for the temporary. Otherwise, if the lvalue has (possibly cv-qualified) type `std::nullptr_t`, the rvalue result is a null pointer constant (4.10). Otherwise, the value contained in the object indicated by the lvalue is the rvalue result.

3 [*Note:* See also 3.10. — *end note*]

4.2 Array-to-pointer conversion [conv.array]

1 An lvalue or rvalue of type "array of `N T`" or "array of unknown bound of `T`" can be converted to an rvalue of type "pointer to `T`". The result is a pointer to the first element of the array.

2 A string literal (2.13.4) with no prefix, with a `u` prefix, with a `U` prefix, or with an `L` prefix can be converted to an rvalue of type "pointer to `char`", "pointer to `char16_t`", "pointer to `char32_t`", or "pointer to `wchar_t`", respectively. In any case, the result is a pointer to the first element of the array. This conversion is considered only when there is an explicit appropriate pointer target type, and not when there is a general need to convert from an lvalue to an rvalue. [*Note:* this conversion is deprecated. See Annex D. — *end note*] For the purpose of ranking in overload resolution (13.3.3.1.1), this conversion is considered an array-to-pointer conversion followed by a qualification conversion (4.4). [*Example:* `"abc"` is converted to "pointer to `const char`" as an array-to-pointer conversion, and then to "pointer to `char`" as a qualification conversion. — *end example*]

4.3 Function-to-pointer conversion [conv.func]

1 An lvalue of function type `T` can be converted to an rvalue of type "pointer to `T` ." The result is a pointer to the function.[52]

2 [*Note:* See 13.4 for additional rules for the case where the function is overloaded. — *end note*]

4.4 Qualification conversions [conv.qual]

1 An rvalue of type "pointer to *cv1* `T` " can be converted to an rvalue of type "pointer to *cv2* `T` " if "*cv2* `T` " is more cv-qualified than "*cv1* `T` ."

2 An rvalue of type "pointer to member of `X` of type *cv1* `T` " can be converted to an rvalue of type "pointer to member of `X` of type *cv2* `T` " if "*cv2* `T` " is more cv-qualified than "*cv1* `T` ."

3 [*Note:* Function types (including those used in pointer to member function types) are never cv-qualified (8.3.5). — *end note*]

4 A conversion can add cv-qualifiers at levels other than the first in multi-level pointers, subject to the following rules:[53]

51) In C++ class rvalues can have cv-qualified types (because they are objects). This differs from ISO C, in which non-lvalues never have cv-qualified types.

52) This conversion never applies to non-static member functions because an lvalue that refers to a non-static member function cannot be obtained.

53) These rules ensure that const-safety is preserved by the conversion.

Two pointer types T1 and T2 are *similar* if there exists a type T and integer $n > 0$ such that:

T1 is $cv_{1,0}$ pointer to $cv_{1,1}$ pointer to $\cdots$ $cv_{1,n-1}$ pointer to $cv_{1,n}$ T

and

T2 is $cv_{2,0}$ pointer to $cv_{2,1}$ pointer to $\cdots$ $cv_{2,n-1}$ pointer to $cv_{2,n}$ T

where each $cv_{i,j}$ is `const`, `volatile`, `const volatile`, or nothing. The n-tuple of cv-qualifiers after the first in a pointer type, e.g., $cv_{1,1}$, $cv_{1,2}$, $\cdots$, $cv_{1,n}$ in the pointer type *T1*, is called the *cv-qualification signature* of the pointer type. An expression of type *T1* can be converted to type *T2* if and only if the following conditions are satisfied:

— the pointer types are similar.

— for every $j > 0$, if `const` is in $cv_{1,j}$ then `const` is in $cv_{2,j}$, and similarly for `volatile`.

— if the $cv_{1,j}$ and $cv_{2,j}$ are different, then `const` is in every $cv_{2,k}$ for $0 < k < j$.

[*Note:* if a program could assign a pointer of type `T**` to a pointer of type `const T**` (that is, if line `//1` below were allowed), a program could inadvertently modify a `const` object (as it is done on line `//2`). For example,

```
int main() {
  const char c = 'c';
  char* pc;
  const char** pcc = &pc;       // 1: not allowed
  *pcc = &c;
  *pc = 'C';                    // 2: modifies a const object
}
```

— *end note*]

5 A *multi-level* pointer to member type, or a *multi-level mixed* pointer and pointer to member type has the form:

cv_0P_0 to cv_1P_1 to $\cdots$ $cv_{n-1}P_{n-1}$ to cv_n T

where P_i is either a pointer or pointer to member and where T is not a pointer type or pointer to member type.

6 Two multi-level pointer to member types or two multi-level mixed pointer and pointer to member types T1 and T2 are *similar* if there exists a type T and integer $n > 0$ such that:

T1 is $cv_{1,0}P_0$ to $cv_{1,1}P_1$ to $\cdots$ $cv_{1,n-1}P_{n-1}$ to $cv_{1,n}$ T

and

T2 is $cv_{2,0}P_0$ to $cv_{2,1}P_1$ to $\cdots$ $cv_{2,n-1}P_{n-1}$ to $cv_{2,n}$ T

7 For similar multi-level pointer to member types and similar multi-level mixed pointer and pointer to member types, the rules for adding cv-qualifiers are the same as those used for similar pointer types.

4.5 Integral promotions [conv.prom]

1 An rvalue of an integer type other than `bool`, `char16_t`, `char32_t`, or `wchar_t` whose integer conversion rank (4.13) is less than the rank of `int` can be converted to an rvalue of type `int` if `int` can represent all the values of the source type; otherwise, the source rvalue can be converted to an rvalue of type `unsigned int`.

2 An rvalue of type `char16_t`, `char32_t`, or `wchar_t` (3.9.1) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, or `unsigned long long int`. If none of the types in that list can represent all the values of its underlying type, an rvalue of type `char16_t`, `char32_t`, or `wchar_t` can be converted to an rvalue of its underlying type. An rvalue of an unscoped enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of the enumeration (i.e. the values in the range b_{min} to b_{max} as described in 7.2): `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, or `unsigned long long int`. If none of the types in that list can represent all the values of the enumeration, an rvalue of an unscoped enumeration type can be converted to an rvalue of the extended integer type with lowest integer conversion rank (4.13) greater than the rank of `long long` in which all the values of the enumeration can be represented. If there are two such extended types, the signed one is chosen.

3 An rvalue for an integral bit-field (9.6) can be converted to an rvalue of type `int` if `int` can represent all the values of the bit-field; otherwise, it can be converted to `unsigned int` if `unsigned int` can represent all the values of the bit-field. If the bit-field is larger yet, no integral promotion applies to it. If the bit-field has an enumerated type, it is treated as any other value of that type for promotion purposes.

4 An rvalue of type `bool` can be converted to an rvalue of type `int`, with `false` becoming zero and `true` becoming one.

5 These conversions are called *integral promotions*.

4.6 Floating point promotion [conv.fpprom]

1 An rvalue of type `float` can be converted to an rvalue of type `double`. The value is unchanged.

2 This conversion is called *floating point promotion*.

4.7 Integral conversions [conv.integral]

1 An rvalue of an integer type can be converted to an rvalue of another integer type. An rvalue of an unscoped enumeration type can be converted to an rvalue of an integer type.

2 If the destination type is unsigned, the resulting value is the least unsigned integer congruent to the source integer (modulo 2^n where n is the number of bits used to represent the unsigned type). [*Note:* In a two's complement representation, this conversion is conceptual and there is no change in the bit pattern (if there is no truncation). — *end note*]

3 If the destination type is signed, the value is unchanged if it can be represented in the destination type (and bit-field width); otherwise, the value is implementation-defined.

4 If the destination type is `bool`, see 4.12. If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

5 The conversions allowed as integral promotions are excluded from the set of integral conversions.

4.8 Floating point conversions [conv.double]

1 An rvalue of floating point type can be converted to an rvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.

2 The conversions allowed as floating point promotions are excluded from the set of floating point conversions.

4.9 Floating-integral conversions [conv.fpint]

1 An rvalue of a floating point type can be converted to an rvalue of an integer type. The conversion truncates; that is, the fractional part is discarded. The behavior is undefined if the truncated value cannot be represented in the destination type. [*Note:* If the destination type is `bool`, see 4.12. — *end note*]

2 An rvalue of an integer type or of an unscoped enumeration type can be converted to an rvalue of a floating point type. The result is exact if possible. Otherwise, it is an implementation-defined choice of either the next lower or higher representable value. [*Note:* loss of precision occurs if the integral value cannot be represented exactly as a value of the floating type. — *end note*] If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to one.

4.10 Pointer conversions [conv.ptr]

1 A *null pointer constant* is an integral constant expression (5.19) rvalue of integer type that evaluates to zero or an rvalue of type `std::nullptr_t`. A null pointer constant can be converted to a pointer type; the result is the *null pointer value* of that type and is distinguishable from every other value of pointer to object or pointer to function type. Two null pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to cv-qualified type is a single conversion, and not the sequence of a pointer conversion followed by a qualification conversion (4.4). A null pointer constant of integral type can be converted to an rvalue of type `std::nullptr_t`. [*Note:* The resulting rvalue is not a null pointer value. — *end note*]

2 An rvalue of type "pointer to *cv* `T` ," where `T` is an effective object type, can be converted to an rvalue of type "pointer to *cv* `void`". The result of converting a "pointer to *cv* `T`" to a "pointer to *cv* `void`" points to the start of the storage location where the object of type `T` resides, as if the object is a most derived object (1.8) of type `T` (that is, not a base class subobject). The null pointer value is converted to the null pointer value of the destination type.

3 An rvalue of type "pointer to *cv* `D`", where `D` is a class type, can be converted to an rvalue of type "pointer to *cv* `B`", where `B` is a base class (Clause 10) of `D`. If `B` is an inaccessible (Clause 11) or ambiguous (10.2) base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion is a pointer to the base class subobject of the derived class object. The null pointer value is converted to the null pointer value of the destination type.

4.11 Pointer to member conversions [conv.mem]

1 A null pointer constant (4.10) can be converted to a pointer to member type; the result is the *null member pointer value* of that type and is distinguishable from any pointer to member not created from a null pointer constant. Two null member pointer values of the same type shall compare equal. The conversion of a null pointer constant to a pointer to member of cv-qualified type is a single conversion, and not the sequence of a pointer to member conversion followed by a qualification conversion (4.4).

2 An rvalue of type "pointer to member of `B` of type *cv* `T`", where `B` is a class type, can be converted to an rvalue of type "pointer to member of `D` of type *cv* `T`", where `D` is a derived class (Clause 10) of `B`. If `B` is an inaccessible (Clause 11), ambiguous (10.2), or virtual (10.1) base class of `D`, or a base class of a virtual base class of `D`, a program that necessitates this conversion is ill-formed. The result of the conversion refers to the same member as the pointer to member before the conversion took place, but it refers to the base class member as if it were a member of the derived class. The result refers to the member in `D`'s instance of `B`. Since the result has type "pointer to member of `D` of type *cv* `T`", it can be dereferenced with a `D` object. The

result is the same as if the pointer to member of `B` were dereferenced with the `B` subobject of `D`. The null member pointer value is converted to the null member pointer value of the destination type.[54]

4.12 Boolean conversions [conv.bool]

1 An rvalue of arithmetic, unscoped enumeration, pointer, or pointer to member type can be converted to an rvalue of type `bool`. A zero value, null pointer value, or null member pointer value is converted to `false`; any other value is converted to `true`. An rvalue of type `std::nullptr_t` can be converted to an rvalue of type `bool`; the resulting value is `false`.

4.13 Integer conversion rank [conv.rank]

1 Every integer type has an *integer conversion rank* defined as follows:

- No two signed integer types shall have the same rank, even if they have the same represesentation.
- The rank of a signed integer type shall be greater than the rank of any signed integer type with a smaller size.
- The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.
- The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type.
- The rank of any standard integer type shall be greater than the rank of any extended integer type with the same size.
- the rank of `char` shall equal the rank of `signed char` and `unsigned char`.
- The rank of `bool` shall be less than the rank of all other standard integer types.
- The ranks of `char16_t`, `char32_t`, and `wchar_t` shall equal the ranks of their underlying types (3.9.1).
- The rank of any extended signed integer type relative to another extended signed integer type with the same size is implementation-defined, but still subject to the other rules for determining the integer conversion rank.
- For all integer types `T1`, `T2`, and `T3`, if `T1` has greater rank than `T2` and `T2` has greater rank than `T3`, then `T1` shall have greater rank than `T3`.

[*Note:* The integer conversion rank is used in the definition of the integral promotions (4.5) and the usual arithmetic conversions (5). — *end note*]

54) The rule for conversion of pointers to members (from pointer to member of base to pointer to member of derived) appears inverted compared to the rule for pointers to objects (from pointer to derived to pointer to base) (4.10, Clause 10). This inversion is necessary to ensure type safety. Note that a pointer to member is not a pointer to object or a pointer to function and the rules for conversions of such pointers do not apply to pointers to members. In particular, a pointer to member cannot be converted to a `void*`.

5 Expressions [expr]

1 [*Note:* Clause 5 defines the syntax, order of evaluation, and meaning of expressions.[55] An expression is a sequence of operators and operands that specifies a computation. An expression can result in a value and can cause side effects. — *end note*]

2 [*Note:* Operators can be overloaded, that is, given meaning when applied to expressions of class type (Clause 9) or enumeration type (7.2). Uses of overloaded operators are transformed into function calls as described in 13.5. Overloaded operators obey the rules for syntax specified in Clause 5, but the requirements of operand type, lvalue, and evaluation order are replaced by the rules for function call. Relations between operators, such as `++a` meaning `a+=1`, are not guaranteed for overloaded operators (13.5), and are not guaranteed for operands of type `bool`. — *end note*]

3 Clause 5 defines the effects of operators when applied to types for which they have not been overloaded. Operator overloading shall not modify the rules for the *built-in operators*, that is, for operators applied to types for which they are defined by this Standard. However, these built-in operators participate in overload resolution, and as part of that process user-defined conversions will be considered where necessary to convert the operands to types appropriate for the built-in operator. If a built-in operator is selected, such conversions will be applied to the operands before the operation is considered further according to the rules in Clause 5; see 13.3.1.2, 13.6.

4 If during the evaluation of an expression, the result is not mathematically defined or not in the range of representable values for its type, the behavior is undefined, unless such an expression appears where an integral constant expression is required (5.19), in which case the program is ill-formed. [*Note:* most existing implementations of C++ ignore integer overflows. Treatment of division by zero, forming a remainder using a zero divisor, and all floating point exceptions vary among machines, and is usually adjustable by a library function. — *end note*]

5 If an expression initially has the type "lvalue reference to `T`" (8.3.2, 8.5.3), the type is adjusted to `T` prior to any further analysis, the expression designates the object or function denoted by the lvalue reference, and the expression is an lvalue.

6 If an expression initially has the type "rvalue reference to T" (8.3.2, 8.5.3), the type is adjusted to "T" prior to any further analysis, and the expression designates the object or function denoted by the rvalue reference. If the expression is the result of calling a function, whether implicitly or explicitly, it is an rvalue; otherwise, it is an lvalue. [*Note:* In general, the effect of this rule is that named rvalue references are treated as lvalues and unnamed rvalue references are treated as rvalues. — *end note*]

[*Example:*

```
struct A { };
A&& operator+(A, A);
A&& f();

A a;
A&& ar = a;
```

The expressions `f()` and `a + a` are rvalues of type `A`. The expression `ar` is an lvalue of type `A`. — *end example*]

55) The precedence of operators is not directly specified, but it can be derived from the syntax.

7 An expression designating an object is called an *object-expression*.

8 In some contexts, *unevaluated operands* appear (5.2.8, 5.3.3, 7.1.6.2). An unevaluated operand is not evaluated. [*Note:* In an unevaluated operand, a non-static class member may be named (5.1) and naming of objects or functions does not, by itself, require that a definition be provided (3.2). — *end note*]

9 Whenever an lvalue expression appears as an operand of an operator that expects an rvalue for that operand, the lvalue-to-rvalue (4.1), array-to-pointer (4.2), or function-to-pointer (4.3) standard conversions are applied to convert the expression to an rvalue. [*Note:* because cv-qualifiers are removed from the type of an expression of non-class type when the expression is converted to an rvalue, an lvalue expression of type `const int` can, for example, be used where an rvalue expression of type `int` is required. — *end note*]

10 Many binary operators that expect operands of arithmetic or enumeration type cause conversions and yield result types in a similar way. The purpose is to yield a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*, which are defined as follows:

— If either operand is of type `long double`, the other shall be converted to `long double`.

— Otherwise, if either operand is `double`, the other shall be converted to `double`.

— Otherwise, if either operand is `float`, the other shall be converted to `float`.

— Otherwise, the integral promotions (4.5) shall be performed on both operands.[56] Then the following rules shall be applied to the promoted operands:

— If both operands have the same type, no further conversion is needed.

— Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank shall be converted to the type of the operand with greater rank.

— Otherwise, if the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type shall be converted to the type of the operand with unsigned integer type.

— Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type shall be converted to the type of the operand with signed integer type.

— Otherwise, both operands shall be converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

11 The values of the floating operands and the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby.[57]

5.1 Primary expressions [expr.prim]

1 Primary expressions are literals, names, names qualified by the scope resolution operator `::`, and lambda expressions.

56) As a consequence, operands of type `bool`, `char16_t`, `char32_t`, `wchar_t`, or an enumerated type are converted to some integral type.

57) The cast and assignment operators must still perform their specific conversions as described in 5.4, 5.2.9 and 5.17.

primary-expression:
literal
`this`
`(` *expression* `)`
id-expression
lambda-expression
id-expression:
unqualified-id
qualified-id
unqualified-id:
identifier
operator-function-id
conversion-function-id
literal-operator-id
~ *class-name*
template-id

2 A *literal* is a primary expression. Its type depends on its form (2.13). A string literal is an lvalue; all other literals are rvalues.

3 The keyword `this` names a pointer to the object for which a non-static member function (9.3.2) is invoked or a non-static data member's initializer (9.2) is evaluated. The keyword `this` shall be used only inside a non-static class member function body (9.3) or in a *brace-or-equal-initializer* for a non-static data member. The type of the expression is a pointer to the class of the function or non-static data member, possibly with cv-qualifiers on the class type. The expression is an rvalue.

4 The operator `::` followed by an *identifier*, a *qualified-id*, or an *operator-function-id* is a *primary-expression*. Its type is specified by the declaration of the identifier, *qualified-id*, or *operator-function-id*. The result is the entity denoted by the identifier, *qualified-id*, or *operator-function-id*. The result is an lvalue if the entity is a function or variable. The identifier, *qualified-id*, or *operator-function-id* shall have global namespace scope or be visible in global scope because of a *using-directive* (7.3.4). [*Note:* the use of `::` allows a type, an object, a function, an enumerator, or a namespace declared in the global namespace to be referred to even if its identifier has been hidden (3.4.3). — *end note*]

5 A parenthesized expression is a primary expression whose type and value are identical to those of the enclosed expression. The presence of parentheses does not affect whether the expression is an lvalue. The parenthesized expression can be used in exactly the same contexts as those where the enclosed expression can be used, and with the same meaning, except as otherwise indicated.

6 An *id-expression* is a restricted form of a *primary-expression*. [*Note:* an *id-expression* can appear after `.` and `->` operators (5.2.5). — *end note*]

7 An *identifier* is an *id-expression* provided it has been suitably declared (Clause 7). [*Note:* for *operator-function-id*s, see 13.5; for *conversion-function-id*s, see 12.3.2; for *literal-operator-id*s, see 13.5.8; for *template-id*s, see 14.2. A *class-name* prefixed by ~ denotes a destructor; see 12.4. Within the definition of a non-static member function, an *identifier* that names a non-static member is transformed to a class member access expression (9.3.1). — *end note*] The type of the expression is the type of the *identifier*. The result is the entity denoted by the identifier. The result is an lvalue if the entity is a function, variable, or data member.

qualified-id:
`::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id*
`::` *identifier*
`::` *operator-function-id*
`::` *template-id*

nested-name-specifier:
type-name `::`
namespace-name `::`
nested-name-specifier identifier `::`
nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`
nested-name-specifier$_{opt}$ *concept-id* `::`

A *nested-name-specifier* that names a class, optionally followed by the keyword `template` (14.2), and then followed by the name of a member of either that class (9.2) or one of its base classes (Clause 10), is a *qualified-id*; 3.4.3.1 describes name lookup for class members that appear in *qualified-ids*. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a static member function or a data member. [*Note:* a class member can be referred to using a *qualified-id* at any point in its potential scope (3.3.6). — *end note*] Where *class-name* `::` *class-name* is used, and the two *class-name*s refer to the same class, this notation names the constructor (12.1). Where *class-name* `::~` *class-name* is used, the two *class-name*s shall refer to the same class; this notation names the destructor (12.4). [*Note:* a *typedef-name* that names a class is a *class-name* (9.1). — *end note*]

8 A *nested-name-specifier* that names a namespace (7.3), followed by the name of a member of that namespace (or the name of a member of a namespace made visible by a *using-directive*) is a *qualified-id*; 3.4.3.2 describes name lookup for namespace members that appear in *qualified-ids*. The result is the member. The type of the result is the type of the member. The result is an lvalue if the member is a function or a variable.

9 A *nested-name-specifier* that names an enumeration (7.2), followed by the name of an enumerator of that enumeration, is a *qualified-id* that refers to the enumerator. The result is the enumerator. The type of the result is the type of the enumeration. The result is an rvalue.

10 In a *qualified-id*, if the *id-expression* is a *conversion-function-id*, its *conversion-type-id* shall denote the same type in both the context in which the entire *qualified-id* occurs and in the context of the class denoted by the *nested-name-specifier*.

11 An *id-expression* that denotes a non-static data member or non-static member function of a class can only be used:

— as part of a class member access (5.2.5) in which the object-expression refers to the member's class or a class derived from that class, or

— to form a pointer to member (5.3.1), or

— in the body of a non-static member function of that class or of a class derived from that class (9.3.1), or

— in a *mem-initializer* for a constructor for that class or for a class derived from that class (12.6.2), or

— in a *brace-or-equal-initializer* for a non-static data member of that class or of a class derived from that class (12.6.2), or

— if that *id-expression* denotes a non-static data member and it is the sole constituent of an unevaluated operand, except for optional enclosing parentheses. [*Example:*

```
struct S {
  int m;
};
int i = sizeof(S::m);           // OK
int j = sizeof(S::m + 42);      // error: reference to non-static member in subexpression
```

— *end example*]

5.1.1 Lambda expressions [expr.prim.lambda]

lambda-expression:
lambda-introducer *lambda-parameter-declaration*$_{opt}$ *compound-statement*

lambda-introducer:
`[` *lambda-capture*$_{opt}$ `]`

lambda-capture:
capture-default
capture-list
capture-default `,` *capture-list*

capture-default:
`&`
`=`

capture-list:
capture
capture-list `,` *capture*

capture:
identifier
`&` *identifier*
`this`

lambda-parameter-declaration:
`(` *lambda-parameter-declaration-list*$_{opt}$ `)` `mutable`$_{opt}$ *attribute-specifier*$_{opt}$
exception-specification$_{opt}$ *lambda-return-type-clause*$_{opt}$

lambda-parameter-declaration-list:
lambda-parameter
lambda-parameter `,` *lambda-parameter-declaration-list*

lambda-parameter:
decl-specifier-seq *attribute-specifier*$_{opt}$ *declarator*

lambda-return-type-clause:
`->` *attribute-specifier*$_{opt}$ *type-id*

1 In a *lambda-parameter-declaration* the *attribute-specifier* appertains to the lambda. In a *lambda-return-type-clause* the attribute appertains to the lambda return type.

2 The evaluation of a *lambda-expression* results in a *closure object*, which is an rvalue. Invoking the closure object executes the statements specified in the *lambda-expression*'s *compound-statement*. Each lambda expression has a unique type. Except as specified below, the type of the closure object is unspecified. [*Note:* A closure object behaves as a function object (20.6) whose function call operator, constructors, and data members are defined by the *lambda-expression* and its context. — *end note*]

3 A name in the *lambda-capture* shall be in scope in the context of the lambda expression, and shall be `this` or shall refer to a local variable or reference with automatic storage duration. [*Note:* A member of an anonymous union is not a variable. — *end note*] The same name shall not appear more than once in a *lambda-capture*. In a *lambda-introducer* of the form `[` *capture-default* `,` *capture-list* `]`, if the *capture-default* is `&`, the *capture-list* shall not contain a *capture* having the prefix `&`, otherwise each *capture* in the *capture-list* other than *this* shall have the prefix `&`.

4 An *effective capture set* is defined as follows:

— For a *lambda-introducer* of the form `[ ]`, the effective capture set is empty.

— For a *lambda-introducer* of the form `[` *capture-list* `]`, the effective capture set consists of the *captures* in the *capture-list*.

— For a *lambda-introducer* of the form `[` *capture-default* `]` or `[` *capture-default* `,` *capture-list* `]`, the effective capture set consists of

— the *capture*s in the *capture-list*, if any, and

— for each name `v` that appears in the lambda expression and denotes a local variable or reference with automatic storage duration in the context where the lambda expression appears and that does not appear in the *capture-list* or as a parameter name in the *lambda-parameter-declaration-list*, `&v` if the *capture-default* is `&` and `v` otherwise, and

— `this` if the lambda expression contains a member access expression referring to `this` (implicitly or explicitly).

5 The *compound-statement* of a lambda expression shall use (3.2) an automatic variable or reference from the context where the lambda expression appears only if the name of the variable or reference is a member of the effective capture set, and shall reference `this` (implicitly or explicitly) only if `this` is a member of the effective capture set. The *compound-statement* of a lambda expression shall not refer to a member of an anonymous union with automatic storage duration.

6 A *lambda-expression* defines a function and the *compound-statement* of a *lambda-expression* has an associated function scope (3.3).

7 The type of the closure object is a class with a unique name, call it `F`, considered to be defined at the point where the lambda expression occurs.

8 Each name `N` in the effective capture set is looked up in the context where the lambda expression appears to determine its object type; in the case of a reference, the object type is the type to which the reference refers. For each element in the effective capture set, `F` has a private non-static data member as follows:

— if the element is `this`, the data member has some unique name, call it `t`, and is of the type of `this` (9.3.2);

— if the element is of the form `& N`, the data member has the name `N` and type "reference to object type of `N`";

— otherwise, the element is of the form `N`, and the data member has the name `N` and type "cv-unqualified object type of `N`".

9 The declaration order of the data members is unspecified.

10 `F` has a public function call operator (13.5.4) with the following properties:

— The *parameter-declaration-clause* is the *lambda-parameter-declaration-list*.

— The return type is the type denoted by the *type-id* in the *lambda-return-type-clause*; for a lambda expression that does not contain a *lambda-return-type-clause* the return type is `void`, unless the *compound-statement* is of the form `{ return` *expression* `; }`, in which case the return type is the type of *expression*.

— The *cv-qualifier-seq* is absent if the lambda expression is `mutable`, and it is `const` otherwise.

— The *exception-specification* is the lambda expression's *exception-specification*, if any.

— The *compound-statement* is obtained from the lambda expression's *compound-statement* as follows: If the lambda expression is within a non-static member function of some class `X`, transform *id-expression*s to class member access syntax as specified in 9.3.1, then replace all occurrences of `this` by `t`. [*Note:* References to captured variables or references within the *compound-statement* refer to the data members of `F`. — *end note*]

11 `F` has an implicitly-declared copy constructor (12.8), and it has a public move constructor that performs a member-wise move. The copy assignment operator in `F` is defined as deleted. The size of `F` is unspecified.

12 If every name in the effective capture set is preceded by `&` and the lambda expression is not `mutable`, `F` is publicly derived from `std::reference_closure<R(P)>` (20.6.18), where `R` is the return type and `P` is the *parameter-type-list* of the lambda expression. Converting an object of type `F` to type `std::reference_closure<R(P)>` and invoking its function call operator shall have the same effect as invoking the function call operator of `F`. [*Note:* This requirement effectively means that such `F`'s must be implemented using a pair of a function pointer and a static scope pointer. — *end note*]

13 The closure object is initialized by direct-initializing each member `N` of `F` with the local variable or reference named `N`; the member `t` is initialized with `this`. If one or more names in the effective capture set are preceded by `&`, the effect of invoking a closure object or a copy after the innermost block scope of the context of the lambda expression has been exited is undefined.

5.2 Postfix expressions [expr.post]

1 Postfix expressions group left-to-right.

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `[` *braced-init-list* `]`
postfix-expression `(` *expression-list*$_{opt}$ `)`
simple-type-specifier `(` *expression-list*$_{opt}$ `)`
typename-specifier `(` *expression-list*$_{opt}$ `)`
simple-type-specifier braced-init-list
typename-specifier braced-init-list
postfix-expression `.` `template`$_{opt}$ *id-expression*
postfix-expression `->` `template`$_{opt}$ *id-expression*
postfix-expression `.` *pseudo-destructor-name*
postfix-expression `->` *pseudo-destructor-name*
postfix-expression `++`
postfix-expression `--`
`dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`typeid` `(` *expression* `)`
`typeid` `(` *type-id* `)`

expression-list:
initializer-list

pseudo-destructor-name:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* `::` $\sim$ *type-name*
`::`$_{opt}$ *nested-name-specifier* `template` *simple-template-id* `::` $\sim$ *type-name*
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ $\sim$ *type-name*

2 [*Note:* The `>` token following the *type-id* in a `dynamic_cast`, `static_cast`, `reinterpret_cast`, or `const_cast` may be the product of replacing a `>>` token by two consecutive `>` tokens (14.2). — *end note*]

3 An *assignment-expression* followed by an ellipsis is a pack expansion (14.5.3).

5.2.1 Subscripting [expr.sub]

1 A postfix expression followed by an expression in square brackets is a postfix expression. One of the expressions shall have the type "pointer to `T`" and the other shall have enumeration or integral type. The result

is an lvalue of type "`T`." The type "`T`" shall be a completely-defined effective object type.[58] The expression `E1[E2]` is identical (by definition) to `*((E1)+(E2))` [*Note:* see 5.3 and 5.7 for details of `*` and `+` and 8.3.4 for details of arrays. — *end note*]

2 A *braced-init-list* may appear as a subscript for a user-defined `operator[]`. In that case, the initializer list is treated as the initializer for the subscript argument of the `operator[]`. An initializer list shall not be used with the built-in subscript operator.

[*Example:*

```
struct X {
  Z operator[](std::initializer_list<int>);
};
X x;
x[{1,2,3}] = 7;           // OK: meaning x.operator[]({1,2,3})
int a[10];
a[{1,2,3}] = 7;           // error: built-in subscript operator
```

— *end example*]

5.2.2 Function call [expr.call]

1 There are two kinds of function call: ordinary function call and member function[59] (9.3) call. A function call is a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the arguments to the function. For an ordinary function call, the postfix expression shall be either an lvalue that refers to a function (in which case the function-to-pointer standard conversion (4.3) is suppressed on the postfix expression), or it shall have pointer to function type. Calling a function through an expression whose function type has a language linkage that is different from the language linkage of the function type of the called function's definition is undefined (7.5). For a member function call, the postfix expression shall be an implicit (9.3.1, 9.4) or explicit class member access (5.2.5) whose *id-expression* is a function member name, or a pointer-to-member expression (5.5) selecting a function member; the call is as a member of the object pointed to or referred to by the object expression (5.2.5, 5.5). In the case of an implicit class member access, the implied object is the one pointed to by `this`. [*Note:* a member function call of the form `f()` is interpreted as `(*this).f()` (see 9.3.1). — *end note*] If a function or member function name is used, the name can be overloaded (Clause 13), in which case the appropriate function shall be selected according to the rules in 13.3. If the selected function is non-virtual, or if the *id-expression* in the class member access expression is a *qualified-id*, that function is called. Otherwise, its final overrider (10.3) in the dynamic type of the object expression is called. [*Note:* the dynamic type is the type of the object pointed or referred to by the current value of the object expression. 12.7 describes the behavior of virtual function calls when the object-expression refers to an object under construction or destruction. — *end note*]

2 [*Note:* if a function or member function name is used, and name lookup (3.4) does not find a declaration of that name, the program is ill-formed. No function is implicitly declared by such a call. — *end note*]

3 The type of the function call expression is the return type of the statically chosen function (i.e., ignoring the `virtual` keyword), even if the type of the function actually called is different. This type shall be a complete effective object type, a reference type or the type `void`.

4 When a function is called, each parameter (8.3.5) shall be initialized (8.5, 12.8, 12.1) with its corresponding argument. If the function is a non-static member function, the `this` parameter of the function (9.3.2) shall be initialized with a pointer to the object of the call, converted as if by an explicit type conversion (5.4).

58) This is true even if the subscript operator is used in the following common idiom: `&x[0]`.
59) A static member function (9.4) is an ordinary function.

[*Note:* There is no access or ambiguity checking on this conversion; the access checking and disambiguation are done as part of the (possibly implicit) class member access operator. See 10.2, 11.2, and 5.2.5. — *end note*] When a function is called, the parameters that have effective object type shall have completely-defined object type. [*Note:* this still allows a parameter to be a pointer or reference to an incomplete class type. However, it prevents a passed-by-value parameter to have an incomplete class type. — *end note*] During the initialization of a parameter, an implementation may avoid the construction of extra temporaries by combining the conversions on the associated argument and/or the construction of temporaries with the initialization of the parameter (see 12.2). The lifetime of a parameter ends when the function in which it is defined returns. The initialization and destruction of each parameter occurs within the context of the calling function. [*Example:* the access of the constructor, conversion functions or destructor is checked at the point of call in the calling function. If a constructor or destructor for a function parameter throws an exception, the search for a handler starts in the scope of the calling function; in particular, if the function called has a *function-try-block* (Clause 15) with a handler that could handle the exception, this handler is not considered. — *end example*] The value of a function call is the value returned by the called function except in a virtual function call if the return type of the final overrider is different from the return type of the statically chosen function, the value returned from the final overrider is converted to the return type of the statically chosen function.

5 [*Note:* a function can change the values of its non-const parameters, but these changes cannot affect the values of the arguments except where a parameter is of a reference type (8.3.2); if the reference is to a const-qualified type, `const_cast` is required to be used to cast away the constness in order to modify the argument's value. Where a parameter is of `const` reference type a temporary object is introduced if needed (7.1.6, 2.13, 2.13.4, 8.3.4, 12.2). In addition, it is possible to modify the values of nonconstant objects through pointer parameters. — *end note*]

6 A function can be declared to accept fewer arguments (by declaring default arguments (8.3.6)) or more arguments (by using the ellipsis, `...`, or a function parameter pack (8.3.5)) than the number of parameters in the function definition (8.4). [*Note:* this implies that, except where the ellipsis (`...`) or a function parameter pack is used, a parameter is available for each argument. — *end note*]

7 When there is no parameter for a given argument, the argument is passed in such a way that the receiving function can obtain the value of the argument by invoking `va_arg` (18.9). The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the argument expression. After these conversions, if the argument does not have arithmetic, enumeration, pointer, pointer to member, or effective class type, the program is ill-formed. Passing a potentially-evaluated argument of class type (Clause 9) with a non-trivial copy constructor or a non-trivial destructor with no corresponding parameter is conditionally-supported, with implementation-defined semantics. If the argument has integral or enumeration type that is subject to the integral promotions (4.5), or a floating point type that is subject to the floating point promotion (4.6), the value of the argument is converted to the promoted type before the call. These promotions are referred to as the *default argument promotions*.

8 [*Note:* The evaluations of the postfix expression and of the argument expressions are all unsequenced relative to one another. All side effects of argument expression evaluations are sequenced before the function is entered (see 1.9). — *end note*]

9 Recursive calls are permitted, except to the function named `main` (3.6.1).

10 A function call is an lvalue if and only if the result type is an lvalue reference.

5.2.3 Explicit type conversion (functional notation) [expr.type.conv]

1 A *simple-type-specifier* (7.1.6.2) or *typename-specifier* (14.6) followed by a parenthesized *expression-list* constructs a value of the specified type given the expression list. If the expression list is a single expression, the type conversion expression is equivalent (in definedness, and if defined in meaning) to the corresponding cast

expression (5.4). If the type specified is a class type, the class type shall be complete. If the expression list specifies more than a single value, the type shall be a class with a suitably declared constructor (8.5, 12.1), and the expression `T(x1, x2, ...)` is equivalent in effect to the declaration `T t(x1, x2, ...);` for some invented temporary variable `t`, with the result being the value of `t` as an rvalue.

2 The expression `T()`, where `T` is a *simple-type-specifier* or *typename-specifier* for a non-array complete effective object type or the (possibly cv-qualified) `void` type, creates an rvalue of the specified type, which is value-initialized (8.5; no initialization is done for the `void()` case). [*Note:* if `T` is a non-class type that is cv-qualified, the *cv-qualifier*s are ignored when determining the type of the resulting rvalue (3.10). —*end note*]

3 Similarly, a *simple-type-specifier* or *typename-specifier* followed by a *braced-init-list* creates a temporary object of the specified type direct-list-initialized (8.5.4) with the specified *braced-init-list*, and its value is that temporary object as an rvalue.

5.2.4 Pseudo destructor call [expr.pseudo]

1 The use of a *pseudo-destructor-name* after a dot `.` or arrow `->` operator represents the destructor for the non-class type named by *type-name*. The result shall only be used as the operand for the function call operator `()`, and the result of such a call has type `void`. The only effect is the evaluation of the *postfix-expression* before the dot or arrow.

2 The left-hand side of the dot operator shall be of scalar type. The left-hand side of the arrow operator shall be of pointer to scalar type. This scalar type is the object type. The *cv*-unqualified versions of the object type and of the type designated by the *pseudo-destructor-name* shall be the same type. Furthermore, the two *type-name*s in a *pseudo-destructor-name* of the form

$$\texttt{::}_{opt}\ \textit{nested-name-specifier}_{opt}\ \textit{type-name}\ \texttt{::}\sim\ \textit{type-name}$$

shall designate the same scalar type.

5.2.5 Class member access [expr.ref]

1 A postfix expression followed by a dot `.` or an arrow `->`, optionally followed by the keyword `template` (14.8.1), and then followed by an *id-expression*, is a postfix expression. The postfix expression before the dot or arrow is evaluated;[60] the result of that evaluation, together with the *id-expression*, determine the result of the entire postfix expression.

2 For the first option (dot) the type of the first expression (the *object expression*) shall be "class object" (of a complete type). For the second option (arrow) the type of the first expression (the *pointer expression*) shall be "pointer to class object" (of a complete type). In these cases, the *id-expression* shall name a member of the class or of one of its base classes. [*Note:* because the name of a class is inserted in its class scope (Clause 9), the name of a class is also considered a nested member of that class. —*end note*] [*Note:* 3.4.5 describes how names are looked up after the `.` and `->` operators. —*end note*]

3 If `E1` has the type "pointer to class `X`," then the expression `E1->E2` is converted to the equivalent form `(*(E1)).E2`; the remainder of 5.2.5 will address only the first option (dot)[61]. Abbreviating *object-expression.id-expression* as `E1.E2`, then the type and lvalue properties of this expression are determined as follows. In the remainder of 5.2.5, *cq* represents either `const` or the absence of `const` *vq* represents either `volatile` or the absence of `volatile`. *cv* represents an arbitrary set of cv-qualifiers, as defined in 3.9.3.

4 If `E2` is declared to have type "reference to `T`," then `E1.E2` is an lvalue; the type of `E1.E2` is `T`. Otherwise, one of the following rules applies.

60) This evaluation happens even if the result is unnecessary to determine the value of the entire postfix expression, for example if the *id-expression* denotes a static member.

61) Note that if `E1` has the type "pointer to class `X`," then `(*(E1))` is an lvalue.

— If E2 is a static data member, and the type of E2 is T, then E1.E2 is an lvalue; the expression designates the named member of the class. The type of E1.E2 is T.

— If E2 is a non-static data member, and the type of E1 is "*cq1 vq1* X", and the type of E2 is "*cq2 vq2* T", the expression designates the named member of the object designated by the first expression. If E1 is an lvalue, then E1.E2 is an lvalue; otherwise, it is an rvalue. Let the notation *vq12* stand for the "union" of *vq1* and *vq2*; that is, if *vq1* or *vq2* is volatile, then *vq12* is volatile. Similarly, let the notation *cq12* stand for the "union" of *cq1* and *cq2*; that is, if *cq1* or *cq2* is const, then *cq12* is const. If E2 is declared to be a mutable member, then the type of E1.E2 is "*vq12* T". If E2 is not declared to be a mutable member, then the type of E1.E2 is "*cq12 vq12* T".

— If E2 is a (possibly overloaded) member function, function overload resolution (13.3) is used to determine whether E1.E2 refers to a static or a non-static member function.

  — If it refers to a static member function, and the type of E2 is "function of parameter-type-list returning T", then E1.E2 is an lvalue; the expression designates the static member function. The type of E1.E2 is the same type as that of E2, namely "function of parameter-type-list returning T".

  — Otherwise, if E1.E2 refers to a non-static member function, and the type of E2 is "function of parameter-type-list *cv* returning T", then E1.E2 is *not* an lvalue. The expression designates a non-static member function. The expression can be used only as the left-hand operand of a member function call (9.3). [*Note:* any redundant set of parentheses surrounding the expression is ignored (5.1). — *end note*] The type of E1.E2 is "function of parameter-type-list *cv* returning T".

— If E2 is a nested type, the expression E1.E2 is ill-formed.

— If E2 is a member enumerator, and the type of E2 is T, the expression E1.E2 is not an lvalue. The type of E1.E2 is T.

5 If E2 is a non-static data member or a non-static member function, the program is ill-formed if the class of which E2 is directly a member is an ambiguous base (10.2) of the naming class (11.2) of E2.

5.2.6 Increment and decrement [expr.post.incr]

1 The value of a postfix ++ expression is the value of its operand. [*Note:* the value obtained is a copy of the original value — *end note*] The operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a complete effective object type. The value of the operand object is modified by adding 1 to it, unless the object is of type bool, in which case it is set to true. [*Note:* this use is deprecated, see Annex D. — *end note*] The value computation of the ++ expression is sequenced before the modification of the operand object. With respect to an indeterminately-sequenced function call, the operation of postfix ++ is a single evaluation. [*Note:* Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single postfix ++ operator. — *end note*] The result is an rvalue. The type of the result is the cv-unqualified version of the type of the operand. See also 5.7 and 5.17.

The operand of postfix -- is decremented analogously to the postfix ++ operator, except that the operand shall not be of type bool. [*Note:* For prefix increment and decrement, see 5.3.2. — *end note*]

5.2.7 Dynamic cast [expr.dynamic.cast]

1 The result of the expression dynamic_cast<T>(v) is the result of converting the expression v to type T. T shall be a pointer or reference to a complete effective class type, or "pointer to *cv* void." The dynamic_cast operator shall not cast away constness (5.2.11).

2 If T is a pointer type, v shall be an rvalue of a pointer to complete effective class type, and the result is an rvalue of type T. If T is an lvalue reference type, v shall be an lvalue of a complete effective class type, and the result is an lvalue of the type referred to by T. If T is an rvalue reference type, v shall be an expression having a complete effective class type, and the result is an rvalue of the type referred to by T.

3 If the type of v is the same as the required result type (which, for convenience, will be called R in this description), or it is the same as R except that the class object type in R is more cv-qualified than the class object type in v, the result is v (converted if necessary).

4 If the value of v is a null pointer value in the pointer case, the result is the null pointer value of type R.

5 If T is "pointer to *cv1* B" and v has type "pointer to *cv2* D" such that B is a base class of D, the result is a pointer to the unique B subobject of the D object pointed to by v. Similarly, if T is "reference to *cv1* B" and v has type *cv2* D such that B is a base class of D, the result is the unique B subobject of the D object referred to by v. [62] The result is an lvalue if T is an lvalue reference, or an rvalue if T is an rvalue reference. In both the pointer and reference cases, *cv1* shall be the same cv-qualification as, or greater cv-qualification than, *cv2*, and B shall be an accessible unambiguous base class of D. [*Example:*

```
struct B { };
struct D : B { };
void foo(D* dp) {
  B*  bp = dynamic_cast<B*>(dp);     // equivalent to B* bp = dp;
}
```

— *end example*]

6 Otherwise, v shall be a pointer to or an lvalue of a polymorphic type (10.3).

7 If T is "pointer to *cv* void," then the result is a pointer to the most derived object pointed to by v. Otherwise, a run-time check is applied to see if the object pointed or referred to by v can be converted to the type pointed or referred to by T.

8 The run-time check logically executes as follows:

— If, in the most derived object pointed (referred) to by v, v points (refers) to a public base class subobject of a T object, and if only one object of type T is derived from the subobject pointed (referred) to by v the result is a pointer (an lvalue referring) to that T object.

— Otherwise, if v points (refers) to a public base class subobject of the most derived object, and the type of the most derived object has a base class, of type T, that is unambiguous and public, the result is a pointer (an lvalue referring) to the T subobject of the most derived object.

— Otherwise, the run-time check *fails*.

9 The value of a failed cast to pointer type is the null pointer value of the required result type. A failed cast to reference type throws std::bad_cast (18.6.3).

[*Example:*

```
class A { virtual void f(); };
class B { virtual void g(); };
class D : public virtual A, private B { };
void g() {
  D   d;
  B*  bp = (B*)&d;                 // cast needed to break protection
  A*  ap = &d;                     // public derivation, no cast needed
  D&  dr = dynamic_cast<D&>(*bp);  // fails
```

62) The most derived object (1.8) pointed or referred to by v can contain other B objects as base classes, but these are ignored.

```
  ap = dynamic_cast<A*>(bp);          // fails
  bp = dynamic_cast<B*>(ap);          // fails
  ap = dynamic_cast<A*>(&d);          //succeeds
  bp = dynamic_cast<B*>(&d);          //fails
}

class E : public D, public B { };
class F : public E, public D { };
void h() {
  F   f;
  A*  ap  = &f;                       // succeeds: finds unique A
  D*  dp  = dynamic_cast<D*>(ap);     // fails: yields 0
                                      // f has two D subobjects
  E*  ep  = (E*)ap;                   // ill-formed: cast from virtual base
  E*  ep1 = dynamic_cast<E*>(ap);     // succeeds
}
```

— *end example*] [*Note:* 12.7 describes the behavior of a `dynamic_cast` applied to an object under construction or destruction. — *end note*]

5.2.8 Type identification [expr.typeid]

1 The result of a `typeid` expression is an lvalue of static type `const std::type_info` (18.6.1) and dynamic type `const std::type_info` or `const` *name* where *name* is an implementation-defined class derived from `std::type_info` which preserves the behavior described in 18.6.1.[63] The lifetime of the object referred to by the lvalue extends to the end of the program. Whether or not the destructor is called for the `std::type_info` object at the end of the program is unspecified.

2 When `typeid` is applied to an lvalue expression whose type is a polymorphic class type (10.3), the result refers to a `std::type_info` object representing the type of the most derived object (1.8) (that is, the dynamic type) to which the lvalue refers. If the lvalue expression is obtained by applying the unary `*` operator to a pointer[64] and the pointer is a null pointer value (4.10), the `typeid` expression throws the `std::bad_typeid` exception (18.6.4).

3 When `typeid` is applied to an expression other than an lvalue of a polymorphic class type, the result refers to a `std::type_info` object representing the static type of the expression. Lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are not applied to the expression. If the type of the expression is a class type, the class shall be completely-defined. The expression is an unevaluated operand (Clause 5).

4 When `typeid` is applied to a *type-id*, the result refers to a `std::type_info` object representing the type of the *type-id*. If the type of the *type-id* is a reference to a possibly *cv*-qualified type, the result of the `typeid` expression refers to a `std::type_info` object representing the *cv*-unqualified referenced type. If the type of the *type-id* is a class type or a reference to a class type, the class shall be completely-defined.

5 The top-level cv-qualifiers of the lvalue expression or the *type-id* that is the operand of `typeid` are always ignored. [*Example:*

```
class D { ... };
D d1;
const D d2;
```

63) The recommended name for such a class is `extended_type_info`.
64) If `p` is an expression of pointer type, then `*p`, `(*p)`, `*(p)`, `((*p))`, `*((p))`, and so on all meet this requirement.

```
typeid(d1) == typeid(d2);       // yields true
typeid(D)  == typeid(const D);  // yields true
typeid(D)  == typeid(d2);       // yields true
typeid(D)  == typeid(const D&); // yields true
```

— *end example*]

6 If the header `<typeinfo>` (18.6.1) is not included prior to a use of `typeid`, the program is ill-formed.

7 [*Note:* 12.7 describes the behavior of `typeid` applied to an object under construction or destruction. — *end note*]

5.2.9 Static cast [expr.static.cast]

1 The result of the expression `static_cast<T>(v)` is the result of converting the expression `v` to type `T`. If `T` is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue. The `static_cast` operator shall not cast away constness (5.2.11).

2 An lvalue of type "*cv1* `B`," where `B` is a class type, can be cast to type "reference to *cv2* `D`," where `D` is a class derived (Clause 10) from `B`, if a valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (4.10), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is neither a virtual base class of `D` nor a base class of a virtual base class of `D`. The result has type "*cv2* `D`." It is an lvalue if the type cast to is an lvalue reference; otherwise, it is an rvalue. An rvalue of type "*cv1* `B`" may be cast to type "rvalue reference to *cv2* `D`" with the same constraints as for an lvalue of type "*cv1* `B`." The result is an rvalue. If the object of type "*cv1* `B`" is actually a subobject of an object of type `D`, the result refers to the enclosing object of type `D`. Otherwise, the result of the cast is undefined. [*Example:*

```
struct B { };
struct D : public B { };
D d;
B &br = d;

static_cast<D&>(br);            // produces lvalue to the original d object
```

— *end example*]

3 Otherwise, an expression `e` can be explicitly converted to a type `T` using a `static_cast` of the form `static_cast<T>(e)` if the declaration `T t(e);` is well-formed, for some invented temporary variable `t` (8.5). The effect of such an explicit conversion is the same as performing the declaration and initialization and then using the temporary variable as the result of the conversion. The result is an lvalue if `T` is an lvalue reference type (8.3.2), and an rvalue otherwise. The expression `e` is used as an lvalue if and only if the initialization uses it as an lvalue.

4 Otherwise, the `static_cast` shall perform one of the conversions listed below. No other conversion shall be performed explicitly using a `static_cast`.

5 Any expression can be explicitly converted to type *cv* `void`. The expression value is discarded. [*Note:* however, if the value is in a temporary variable (12.2), the destructor for that variable is not executed until the usual time, and the value of the variable is preserved for the purpose of executing the destructor. — *end note*] The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the expression.

6 The inverse of any standard conversion sequence (Clause 4), other than the lvalue-to-rvalue (4.1), array-to-pointer (4.2), function-to-pointer (4.3), and boolean (4.12) conversions, can be performed explicitly using `static_cast`. A program is ill-formed if it uses `static_cast` to perform the inverse of an ill-formed standard conversion sequence. [*Example:*

```
struct B { };
struct D : private B { };
void f() {
  static_cast<D*>((B*)0);              // Error: B is a private base of D.
  static_cast<int B::*>((int D::*)0);  // Error: B is a private base of D.
}
```

— *end example*]

7 The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) conversions are applied to the operand. Such a `static_cast` is subject to the restriction that the explicit conversion does not cast away constness (5.2.11), and the following additional rules for specific cases:

8 A value of a scoped enumeration type (7.2) can be explicitly converted to an integral type. The value is unchanged if the original value can be represented by the specified type. Otherwise, the resulting value is unspecified.

9 A value of integral or enumeration type can be explicitly converted to an enumeration type. The value is unchanged if the original value is within the range of the enumeration values (7.2). Otherwise, the resulting enumeration value is unspecified.

10 An rvalue of type "pointer to *cv1* `B`," where `B` is a class type, can be converted to an rvalue of type "pointer to *cv2* `D`," where `D` is a class derived (Clause 10) from `B`, if a valid standard conversion from "pointer to `D`" to "pointer to `B`" exists (4.10), *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*, and `B` is neither a virtual base class of `D` nor a base class of a virtual base class of `D`. The null pointer value (4.10) is converted to the null pointer value of the destination type. If the rvalue of type "pointer to *cv1* `B`" points to a `B` that is actually a subobject of an object of type `D`, the resulting pointer points to the enclosing object of type `D`. Otherwise, the result of the cast is undefined.

11 An rvalue of type "pointer to member of `D` of type *cv1* `T`" can be converted to an rvalue of type "pointer to member of `B`" of type *cv2* `T`, where `B` is a base class (Clause 10) of `D`, if a valid standard conversion from "pointer to member of `B` of type `T`" to "pointer to member of `D` of type `T`" exists (4.11), and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.[65] The null member pointer value (4.11) is converted to the null member pointer value of the destination type. If class `B` contains the original member, or is a base or derived class of the class containing the original member, the resulting pointer to member points to the original member. Otherwise, the result of the cast is undefined. [*Note:* although class `B` need not contain the original member, the dynamic type of the object on which the pointer to member is dereferenced must contain the original member; see 5.5. — *end note*]

12 An rvalue of type "pointer to *cv1* `void`" can be converted to an rvalue of type "pointer to *cv2* `T`," where `T` is an object type and *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*. The null pointer value is converted to the null pointer value of the destination type. A value of type pointer to object converted to "pointer to *cv* `void`" and back, possibly with different cv-qualification, shall have its original value. [*Example:*

```
T* p1 = new T;
const T* p2 = static_cast<const T*>(static_cast<void*>(p1));
bool b = p1 == p2;  // b will have the value true.
```

— *end example*]

5.2.10 Reinterpret cast [expr.reinterpret.cast]

1 The result of the expression `reinterpret_cast<T>(v)` is the result of converting the expression `v` to type

65) Function types (including those used in pointer to member function types) are never cv-qualified; see 8.3.5.

`T`. If `T` is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the the expression `v`. Conversions that can be performed explicitly using `reinterpret_cast` are listed below. No other conversion can be performed explicitly using `reinterpret_cast`.

2 The `reinterpret_cast` operator shall not cast away constness. [*Note:* see 5.2.11 for the definition of "casting away constness". Subject to the restrictions in this section, an expression may be cast to its own type using a `reinterpret_cast` operator. — *end note*]

3 The mapping performed by `reinterpret_cast` is implementation-defined. [*Note:* it might, or might not, produce a representation different from the original value. — *end note*]

4 A pointer can be explicitly converted to any integral type large enough to hold it. The mapping function is implementation-defined. [*Note:* it is intended to be unsurprising to those who know the addressing structure of the underlying machine. — *end note*] A value of type `std::nullptr_t` can be converted to an integral type; the conversion has the same meaning and validity as a conversion of `(void*)0` to the integral type. [*Note:* a `reinterpret_cast` cannot be used to convert a value of any type to the type `std::nullptr_t`. — *end note*]

5 A value of integral type or enumeration type can be explicitly converted to a pointer. A pointer converted to an integer of sufficient size (if any such exists on the implementation) and back to the same pointer type will have its original value; mappings between pointers and integers are otherwise implementation-defined.

6 A pointer to a function can be explicitly converted to a pointer to a function of a different type. The effect of calling a function through a pointer to a function type (8.3.5) that is not the same as the type used in the definition of the function is undefined. Except that converting an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are function types) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified. [*Note:* see also 4.10 for more details of pointer conversions. — *end note*]

7 A pointer to an effective object can be explicitly converted to a pointer to a different effective object type.[66] Except that converting an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" (where `T1` and `T2` are object types and where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer value, the result of such a pointer conversion is unspecified.

8 Converting a pointer to a function into a pointer to an object type or vice versa is conditionally-supported. The meaning of such a conversion is implementation defined, except that if an implementation supports conversions in both directions, converting an rvalue of one type to the other type and back, possibly with different cv-qualification, shall yield the original pointer value.

9 The null pointer value (4.10) is converted to the null pointer value of the destination type. [*Note:* A null pointer constant of type `std::nullptr_t` cannot be converted to a pointer type, and a null pointer constant of integral type is not necessarily converted to a null pointer value. — *end note*]

10 An rvalue of type "pointer to member of `X` of type `T1`" can be explicitly converted to an rvalue of type "pointer to member of `Y` of type `T2`" if `T1` and `T2` are both function types or both effective object types.[67] The null member pointer value (4.11) is converted to the null member pointer value of the destination type. The result of this conversion is unspecified, except in the following cases:

— converting an rvalue of type "pointer to member function" to a different pointer to member function type and back to its original type yields the original pointer to member value.

66) The types may have different cv-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.

67) `T1` and `T2` may have different cv-qualifiers, subject to the overall restriction that a `reinterpret_cast` cannot cast away constness.

— converting an rvalue of type "pointer to data member of `X` of type `T1`" to the type "pointer to data member of `Y` of type `T2`" (where the alignment requirements of `T2` are no stricter than those of `T1`) and back to its original type yields the original pointer to member value.

11 An lvalue expression of type `T1` can be cast to the type "reference to `T2`" if an expression of type "pointer to `T1`" can be explicitly converted to the type "pointer to `T2`" using a `reinterpret_cast`. That is, a reference cast `reinterpret_cast<T&>(x)` has the same effect as the conversion `*reinterpret_cast<T*>(&x)` with the built-in `&` and `*` operators (and similarly for `reinterpret_cast<T&&>(x)`). The result refers to the same object as the source lvalue, but with a different type. The result is an lvalue for lvalue references or an rvalue for rvalue references. No temporary is created, no copy is made, and constructors (12.1) or conversion functions (12.3) are not called.[68]

5.2.11 Const cast [expr.const.cast]

1 The result of the expression `const_cast<T>(v)` is of type `T`. If `T` is an lvalue reference type, the result is an lvalue; otherwise, the result is an rvalue and the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the expression `v`. Conversions that can be performed explicitly using `const_cast` are listed below. No other conversion shall be performed explicitly using `const_cast`.

2 [*Note:* Subject to the restrictions in this section, an expression may be cast to its own type using a `const_cast` operator. — *end note*]

3 For two pointer types `T1` and `T2` where

T1 is $cv_{1,0}$ pointer to $cv_{1,1}$ pointer to $\cdots$ $cv_{1,n-1}$ pointer to $cv_{1,n}$ *T*

and

T2 is $cv_{2,0}$ pointer to $cv_{2,1}$ pointer to $\cdots$ $cv_{2,n-1}$ pointer to $cv_{2,n}$ *T*

where `T` is any effective object type or the `void` type and where $cv_{1,k}$ and $cv_{2,k}$ may be different cv-qualifications, an rvalue of type `T1` may be explicitly converted to the type `T2` using a `const_cast`. The result of a pointer `const_cast` refers to the original object.

4 An lvalue of type `T1` can be explicitly converted to an lvalue of type `T2` using the cast `const_cast<T2&>` (where `T1` and `T2` are effective object types) if a pointer to `T1` can be explicitly converted to the type "pointer to `T2`" using a `const_cast`. Similarly, for two effective object types `T1` and `T2`, an expression of type `T1` can be explicitly converted to an rvalue of type `T2` using the cast `const_cast<T2&&>` if a pointer to `T1` can be explicitly converted to the type "pointer to `T2`" using a `const_cast`. The result of a reference `const_cast` refers to the original object.

5 For a `const_cast` involving pointers to data members, multi-level pointers to data members and multi-level mixed pointers and pointers to data members (4.4), the rules for `const_cast` are the same as those used for pointers; the "member" aspect of a pointer to member is ignored when determining where the cv-qualifiers are added or removed by the `const_cast`. The result of a pointer to data member `const_cast` refers to the same member as the original (uncast) pointer to data member.

6 A null pointer value (4.10) is converted to the null pointer value of the destination type. The null member pointer value (4.11) is converted to the null member pointer value of the destination type.

7 [*Note:* Depending on the type of the object, a write operation through the pointer, lvalue or pointer to data member resulting from a `const_cast` that casts away a const-qualifier[69] may produce undefined behavior (7.1.6.1). — *end note*]

68) This is sometimes referred to as a *type pun*.
69) `const_cast` is not limited to conversions that cast away a const-qualifier.

8 The following rules define the process known as *casting away constness*. In these rules `Tn` and `Xn` represent types. For two pointer types:

`X1` is `T1`$cv_{1,1}$ `*` $\cdots$ $cv_{1,N}$ `*` where `T1` is not a pointer type

`X2` is `T2`$cv_{2,1}$ `*` $\cdots$ $cv_{2,M}$ `*` where `T2` is not a pointer type

K is $\min(N, M)$

casting from `X1` to `X2` casts away constness if, for a non-pointer type `T` there does not exist an implicit conversion (Clause 4) from:

`T`$cv_{1,(N-K+1)}$ `*` $cv_{1,(N-K+2)}$ `*` $\cdots$ $cv_{1,N}$ `*`

to

`T`$cv_{2,(M-K+1)}$ `*` $cv_{2,(M-K+2)}$ `*` $\cdots$ $cv_{2,M}$ `*`

9 Casting from an lvalue of type `T1` to an lvalue of type `T2` using a reference cast casts away constness if a cast from an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" casts away constness.

10 Casting from an rvalue of type "pointer to data member of `X` of type `T1`" to the type "pointer to data member of `Y` of type `T2`" casts away constness if a cast from an rvalue of type "pointer to `T1`" to the type "pointer to `T2`" casts away constness.

11 For multi-level pointer to members and multi-level mixed pointers and pointer to members (4.4), the "member" aspect of a pointer to member level is ignored when determining if a `const` cv-qualifier has been cast away.

12 [*Note:* some conversions which involve only changes in cv-qualification cannot be done using `const_cast`. For instance, conversions between pointers to functions are not covered because such conversions lead to values whose use causes undefined behavior. For the same reasons, conversions between pointers to member functions, and in particular, the conversion from a pointer to a const member function to a pointer to a non-const member function, are not covered. — *end note*]

5.3 Unary expressions [expr.unary]

1 Expressions with unary operators group right-to-left.

unary-expression:
postfix-expression
`++` *cast-expression*
`--` *cast-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
new-expression
delete-expression

unary-operator: one of
`* & + - ! ~`

5.3.1 Unary operators [expr.unary.op]

1 The unary `*` operator performs *indirection*: the expression to which it is applied shall be a pointer to an effective object type, or a pointer to a function type and the result is an lvalue referring to the object or

function to which the expression points. If the type of the expression is "pointer to `T`," the type of the result is "`T`." [*Note:* a pointer to an incomplete type (other than *cv* `void`) can be dereferenced. The lvalue thus obtained can be used in limited ways (to initialize a reference, for example); this lvalue must not be converted to an rvalue, see 4.1. — *end note*]

2 The result of the unary `&` operator is a pointer to its operand. The operand shall be an lvalue or a *qualified-id*. In the first case, if the type of the expression is "`T`," the type of the result is "pointer to `T`." In particular, the address of an object of type "*cv* `T`" is "pointer to *cv* `T`," with the same cv-qualifiers. For a *qualified-id*, if the member is a static member of type "`T`", the type of the result is plain "pointer to `T`." If the member is a non-static member of class `C` of type `T`, the type of the result is "pointer to member of `class C` of type `T`." The address of a member of a concept map (14.9.2) shall not be taken, either implicitly or explicitly, nor shall a member of a concept map be bound to a reference. [*Example:*

```
struct A { int i; };
struct B : A { };
... &B::i ...        // has type int A::*
```

— *end example*] [*Note:* a pointer to member formed from a `mutable` non-static data member (7.1.1) does not reflect the `mutable` specifier associated with the non-static data member. — *end note*]

3 A pointer to member is only formed when an explicit `&` is used and its operand is a *qualified-id* not enclosed in parentheses. [*Note:* that is, the expression `&(qualified-id)`, where the *qualified-id* is enclosed in parentheses, does not form an expression of type "pointer to member." Neither does `qualified-id`, because there is no implicit conversion from a *qualified-id* for a non-static member function to the type "pointer to member function" as there is from an lvalue of function type to the type "pointer to function" (4.3). Nor is `&unqualified-id` a pointer to member, even within the scope of the *unqualified-id*'s class. — *end note*]

4 The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares `operator&()` as a member function, then the behavior is undefined (and no diagnostic is required). The operand of `&` shall not be a bit-field.

5 The address of an overloaded function (Clause 13) can be taken only in a context that uniquely determines which version of the overloaded function is referred to (see 13.4). [*Note:* since the context might determine whether the operand is a static or non-static member function, the context can also affect whether the expression has type "pointer to function" or "pointer to member function." — *end note*]

6 The operand of the unary `+` operator shall have arithmetic, enumeration, or pointer type and the result is the value of the argument. Integral promotion is performed on integral or enumeration operands. The type of the result is the type of the promoted operand.

7 The operand of the unary `-` operator shall have arithmetic or enumeration type and the result is the negation of its operand. Integral promotion is performed on integral or enumeration operands. The negative of an unsigned quantity is computed by subtracting its value from 2^n, where n is the number of bits in the promoted operand. The type of the result is the type of the promoted operand.

8 The operand of the logical negation operator `!` is contextually converted to `bool` (Clause 4); its value is `true` if the converted operand is `false` and `false` otherwise. The type of the result is `bool`.

9 The operand of `~` shall have integral or enumeration type; the result is the one's complement of its operand. Integral promotions are performed. The type of the result is the type of the promoted operand. There is an ambiguity in the *unary-expression* `~X()`, where `X` is a *class-name*. The ambiguity is resolved in favor of treating `~` as a unary complement rather than treating `~X` as referring to a destructor.

5.3.2 Increment and decrement [expr.pre.incr]

1 The operand of prefix `++` is modified by adding `1`, or set to `true` if it is `bool` (this use is deprecated). The

operand shall be a modifiable lvalue. The type of the operand shall be an arithmetic type or a pointer to a completely-defined effective object type. The result is the updated operand; it is an lvalue, and it is a bit-field if the operand is a bit-field. If `x` is not of type `bool`, the expression `++x` is equivalent to `x+=1` [*Note:* see the discussions of addition (5.7) and assignment operators (5.17) for information on conversions. — *end note*]

2 The operand of prefix `--` is modified by subtracting `1`. The operand shall not be of type `bool`. The requirements on the operand of prefix `--` and the properties of its result are otherwise the same as those of prefix `++`. [*Note:* For postfix increment and decrement, see 5.2.6. — *end note*]

5.3.3 Sizeof [expr.sizeof]

1 The `sizeof` operator yields the number of bytes in the object representation of its operand. The operand is either an expression, which is an unevaluated operand (Clause 5), or a parenthesized *type-id*. The `sizeof` operator shall not be applied to an expression that has function or incomplete type, or to an enumeration type before all its enumerators have been declared, or to the parenthesized name of such types, or to an lvalue that designates a bit-field. `sizeof(char)`, `sizeof(signed char)` and `sizeof(unsigned char)` are `1`. The result of `sizeof` applied to any other fundamental type (3.9.1) is implementation-defined. [*Note:* in particular, `sizeof(bool)`, `sizeof(char16_t)`, `sizeof(char32_t)`, and `sizeof(wchar_t)` are implementation-defined.[70] — *end note*] [*Note:* See 1.7 for the definition of *byte* and 3.9 for the definition of *object representation*. — *end note*]

2 When applied to a reference or a reference type, the result is the size of the referenced type. When applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array. The size of a most derived class shall be greater than zero (1.8). The result of applying `sizeof` to a base class subobject is the size of the base class type.[71] When applied to an array, the result is the total number of bytes in the array. This implies that the size of an array of n elements is n times the size of an element.

3 The `sizeof` operator can be applied to a pointer to a function, but shall not be applied directly to a function.

4 The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the operand of `sizeof`.

5 The identifier in a `sizeof...` expression shall name a parameter pack. The `sizeof...` operator yields the number of arguments provided for the parameter pack *identifier*. The parameter pack is expanded (14.5.3) by the `sizeof...` operator. [*Example:*

```
template<class... Types>
struct count {
  static const std::size_t value = sizeof...(Types);
};
```

— *end example*]

6 The result of `sizeof` and `sizeof...` is a constant of type `std::size_t`. [*Note:* `std::size_t` is defined in the standard header `<cstddef>` (18.1). — *end note*]

5.3.4 New [expr.new]

1 The *new-expression* attempts to create an object of the *type-id* (8.1) or *new-type-id* to which it is applied. The type of that object is the *allocated type*. This type shall be a complete object type, but not an

70) `sizeof(bool)` is not required to be `1`.

71) The actual size of a base class subobject may be less than the result of applying `sizeof` to the subobject, due to virtual base classes and less strict padding requirements on base class subobjects.

abstract class type or array thereof (1.8, 3.9, 10.4). It is implementation-defined whether over-aligned types are supported (3.11). [*Note:* because references are not objects, references cannot be created by *new-expressions*. — *end note*] [*Note:* the *type-id* may be a cv-qualified type, in which case the object created by the *new-expression* has a cv-qualified type. — *end note*]

new-expression:
    ::$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
    ::$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
    `(` *expression-list* `)`

new-type-id:
    type-specifier-seq new-declarator$_{opt}$

new-declarator:
    ptr-operator new-declarator$_{opt}$
    noptr-new-declarator

noptr-new-declarator:
    `[` *expression* `]`
    noptr-new-declarator `[` *constant-expression* `]`

new-initializer:
    `(` *expression-list*$_{opt}$ `)`
    braced-init-list

Entities created by a *new-expression* have dynamic storage duration (3.7.4). [*Note:* the lifetime of such an entity is not necessarily restricted to the scope in which it is created. — *end note*] If the entity is a non-array object, the *new-expression* returns a pointer to the object created. If it is an array, the *new-expression* returns a pointer to the initial element of the array.

2 If the `auto` *type-specifier* appears in the *type-specifier-seq* of a *new-type-id* or *type-id* of a *new-expression*, the *new-expression* shall contain a *new-initializer* of the form

    `(` *assignment-expression* `)`

The allocated type is deduced from the *new-initializer* as follows: Let `(e)` be the *new-initializer* and `T` be the *new-type-id* or *type-id* of the *new-expression*, then the allocated type is the type deduced for the variable `x` in the invented declaration (7.1.6.4):

```
T x = e;
```

[*Example:*

```
new auto(1);                    // allocated type is int
auto x = new auto('a');         // allocated type is char, x is of type char*
```

— *end example*]

3 The *new-type-id* in a *new-expression* is the longest possible sequence of *new-declarators*. [*Note:* this prevents ambiguities between declarator operators `&`, `*`, `[]`, and their expression counterparts. — *end note*] [*Example:*

```
new int * i;        // syntax error: parsed as (new int*) i, not as (new int)*i
```

The `*` is the pointer declarator and not the multiplication operator. — *end example*]

4 [*Note:* parentheses in a *new-type-id* of a *new-expression* can have surprising effects. [*Example:*

```
new int(*[10])();               // error
```

is ill-formed because the binding is

```
(new int) (*[10])();              // error
```

Instead, the explicitly parenthesized version of the `new` operator can be used to create objects of compound types (3.9.2):

```
new (int (*[10])());
```

allocates an array of `10` pointers to functions (taking no argument and returning `int`. — *end example*] — *end note*]

5 When the allocated object is an array (that is, the *noptr-new-declarator* syntax is used or the *new-type-id* or *type-id* denotes an array type), the *new-expression* yields a pointer to the initial element (if any) of the array. [*Note:* both `new int` and `new int[10]` have type `int*` and the type of `new int[i][10]` is `int (*)[10]` — *end note*]

6 Every *constant-expression* in a *noptr-new-declarator* shall be an integral constant expression (5.19) and evaluate to a strictly positive value. The *expression* in a *noptr-new-declarator* shall be of integral type, enumeration type, or a class type for which a single non-explicit conversion function to integral or enumeration type exists (12.3). If the expression is of class type, the expression is converted by calling that conversion function, and the result of the conversion is used in place of the original expression. If the value of the expression is negative, the behavior is undefined. [*Example:* given the definition `int n = 42`, `new float[n][5]` is well-formed (because `n` is the *expression* of a *noptr-new-declarator*), but `new float[5][n]` is ill-formed (because `n` is not a constant expression). If `n` is negative, the effect of `new float[n][5]` is undefined. — *end example*]

7 When the value of the *expression* in a *noptr-new-declarator* is zero, the allocation function is called to allocate an array with no elements. If the value of that *expression* is such that the size of the allocated object would exceed the implementation-defined limit, no storage is obtained and the *new-expression* terminates by throwing an exception of a type that would match a handler (15.3) of type `std::length_error` (19.1.4).

8 A *new-expression* obtains storage for the object by calling an *allocation function* (3.7.4.1). If the *new-expression* terminates by throwing an exception, it may release storage by calling a deallocation function (3.7.4.2). If the allocated type is a non-array type, the allocation function's name is `operator new` and the deallocation function's name is `operator delete`. If the allocated type is an array type, the allocation function's name is `operator new[]` and the deallocation function's name is `operator delete[]`. [*Note:* an implementation shall provide default definitions for the global allocation functions (3.7.4, 18.5.1.1, 18.5.1.2). A C++ program can provide alternative definitions of these functions (17.6.4.6) and/or class-specific versions (12.5). — *end note*]

9 If the *new-expression* begins with a unary `::` operator, the allocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or array thereof, the allocation function's name is looked up in the scope of `T` If this lookup fails to find the name, or if the allocated type is not a class type, the allocation function's name is looked up in the global scope.

10 A *new-expression* passes the amount of space requested to the allocation function as the first argument of type `std::size_t`. That argument shall be no less than the size of the object being created; it may be greater than the size of the object being created only if the object is an array. For arrays of `char` and `unsigned char`, the difference between the result of the *new-expression* and the address returned by the allocation function shall be an integral multiple of the strictest fundamental alignment requirement (3.11) of any object type whose size is no greater than the size of the array being created. [*Note:* Because allocation functions are assumed to return pointers to storage that is appropriately aligned for objects of any type with fundamental alignment, this constraint on array allocation overhead permits the common idiom of allocating character arrays into which objects of other types will later be placed. — *end note*]

11 The *new-placement* syntax is used to supply additional arguments to an allocation function. If used, overload resolution is performed on a function call created by assembling an argument list consisting of the amount of space requested (the first argument) and the expressions in the *new-placement* part of the *new-expression* (the second and succeeding arguments). The first of these arguments has type `std::size_t` and the remaining arguments have the corresponding types of the expressions in the *new-placement*.

12 [*Example:*

- `new T` results in a call of `operator new(sizeof(T))`,
- `new(2,f) T` results in a call of `operator new(sizeof(T),2,f)`,
- `new T[5]` results in a call of `operator new[](sizeof(T)*5+x)`, and
- `new(2,f) T[5]` results in a call of `operator new[](sizeof(T)*5+y,2,f)`.

Here, `x` and `y` are non-negative unspecified values representing array allocation overhead; the result of the *new-expression* will be offset by this amount from the value returned by `operator new[]`. This overhead may be applied in all array *new-expression*s, including those referencing the library function `operator new[](std::size_t, void*)` and other placement allocation functions. The amount of overhead may vary from one invocation of `new` to another. — *end example*]

13 [*Note:* unless an allocation function is declared with an empty *exception-specification* (15.4), `throw()`, it indicates failure to allocate storage by throwing a `std::bad_alloc` exception (Clause 15, 18.5.2.1); it returns a non-null pointer otherwise. If the allocation function is declared with an empty *exception-specification*, `throw()`, it returns null to indicate failure to allocate storage and a non-null pointer otherwise. — *end note*] If the allocation function returns null, initialization shall not be done, the deallocation function shall not be called, and the value of the *new-expression* shall be null.

14 [*Note:* when the allocation function returns a value other than null, it must be a pointer to a block of storage in which space for the object has been reserved. The block of storage is assumed to be appropriately aligned and of the requested size. The address of the created object will not necessarily be the same as that of the block if the object is an array. — *end note*]

15 A *new-expression* that creates an object of type `T` initializes that object as follows:

- If the *new-initializer* is omitted, the object is default-initialized (8.5); if no initialization is performed, the object has inteterminate value.
- Otherwise, the *new-initializer* is interpreted according to the initialization rules of 8.5 for direct-initialization.

16 If the *new-expression* creates an object or an array of objects of class type, access and ambiguity control are done for the allocation function, the deallocation function (12.5), and the constructor (12.1). If the new expression creates an array of objects of class type, access and ambiguity control are done for the destructor (12.4).

17 If any part of the object initialization described above[72] terminates by throwing an exception and a suitable deallocation function can be found, the deallocation function is called to free the memory in which the object was being constructed, after which the exception continues to propagate in the context of the *new-expression*. If no unambiguous matching deallocation function can be found, propagating the exception does not cause the object's memory to be freed. [*Note:* This is appropriate when the called allocation function does not allocate memory; otherwise, it is likely to result in a memory leak. — *end note*]

18 If the *new-expression* begins with a unary `::` operator, the deallocation function's name is looked up in the global scope. Otherwise, if the allocated type is a class type `T` or an array thereof, the deallocation function's

72) This may include evaluating a *new-initializer* and/or calling a constructor.

name is looked up in the scope of `T`. If this lookup fails to find the name, or if the allocated type is not a class type or array thereof, the deallocation function's name is looked up in the global scope.

19 A declaration of a placement deallocation function matches the declaration of a placement allocation function if it has the same number of parameters and, after parameter transformations (8.3.5), all parameter types except the first are identical. Any non-placement deallocation function matches a non-placement allocation function. If the lookup finds a single matching deallocation function, that function will be called; otherwise, no deallocation function will be called. If the lookup finds the two-parameter form of a usual deallocation function (3.7.4.2) and that function, considered as a placement deallocation function, would have been selected as a match for the allocation function, the program is ill-formed. [*Example:*

```
struct S {
  // Placement allocation function:
  static void* operator new(std::size_t, std::size_t);

  // Usual (non-placement) deallocation function:
  static void operator delete(void*, std::size_t);
};

S* p = new (0) S;    // ill-formed: non-placement deallocation function matches
                     // placement allocation function
```

— *end example*]

20 If a *new-expression* calls a deallocation function, it passes the value returned from the allocation function call as the first argument of type `void*`. If a placement deallocation function is called, it is passed the same additional arguments as were passed to the placement allocation function, that is, the same arguments as those specified with the *new-placement* syntax. If the implementation is allowed to make a copy of any argument as part of the call to the allocation function, it is allowed to make a copy (of the same original value) as part of the call to the deallocation function or to reuse the copy made as part of the call to the allocation function. If the copy is elided in one place, it need not be elided in the other.

21 Whether the allocation function is called before evaluating the constructor arguments or after evaluating the constructor arguments but before entering the constructor is unspecified. It is also unspecified whether the arguments to a constructor are evaluated if the allocation function returns the null pointer or exits using an exception.

5.3.5 Delete [expr.delete]

1 The *delete-expression* operator destroys a most derived object (1.8) or array created by a *new-expression*.

delete-expression:
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*

The first alternative is for non-array objects, and the second is for arrays. Whenever the `delete` keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative.[73] The operand shall have a pointer type, or a class type having a single non-explicit conversion function (12.3.2) to a pointer type. The result has type `void`.

2 If the operand has a class type, the operand is converted to a pointer type by calling the above-mentioned conversion function, and the converted operand is used in place of the original operand for the remainder of this section. In either alternative, the value of the operand of `delete` may be a null pointer value. If it is

73) A lambda expression with a *lambda-introducer* that consists of empty square brackets can follow the `delete` keyword if the lambda expression is enclosed in parentheses.

not a null pointer value, in the first alternative (*delete object*), the value of the operand of `delete` shall be a pointer to a non-array object or a pointer to a subobject (1.8) representing a base class of such an object (Clause 10). If not, the behavior is undefined. In the second alternative (*delete array*), the value of the operand of `delete` shall be the pointer value which resulted from a previous array *new-expression*.[74] If not, the behavior is undefined. [*Note:* this means that the syntax of the *delete-expression* must match the type of the object allocated by new, not the syntax of the *new-expression*. — *end note*] [*Note:* a pointer to a `const` type can be the operand of a *delete-expression*; it is not necessary to cast away the constness (5.2.11) of the pointer expression before it is used as the operand of the *delete-expression*. — *end note*]

3 In the first alternative (*delete object*), if the static type of the object to be deleted is different from its dynamic type, the static type shall be a base class of the dynamic type of the object to be deleted and the static type shall have a virtual destructor or the behavior is undefined. In the second alternative (*delete array*) if the dynamic type of the object to be deleted differs from its static type, the behavior is undefined.[75]

4 The *cast-expression* in a *delete-expression* shall be evaluated exactly once.

5 If the object being deleted has incomplete class type at the point of deletion and the complete class has a non-trivial destructor or a deallocation function, the behavior is undefined.

6 If the value of the operand of the *delete-expression* is not a null pointer value, the *delete-expression* will invoke the destructor (if any) for the object or the elements of the array being deleted. In the case of an array, the elements will be destroyed in order of decreasing address (that is, in reverse order of the completion of their constructor; see 12.6.2).

7 If the value of the operand of the *delete-expression* is not a null pointer value, the *delete-expression* will call a *deallocation function* (3.7.4.2). Otherwise, it is unspecified whether the deallocation function will be called. [*Note:* The deallocation function is called regardless of whether the destructor for the object or some element of the array throws an exception. — *end note*]

8 [*Note:* An implementation provides default definitions of the global deallocation functions `operator delete()` for non-arrays (18.5.1.1) and `operator delete[]()` for arrays (18.5.1.2). A C++ program can provide alternative definitions of these functions (17.6.4.6), and/or class-specific versions (12.5). — *end note*] When the keyword `delete` in a *delete-expression* is preceded by the unary `::` operator, the global deallocation function is used to deallocate the storage.

9 Access and ambiguity control are done for both the deallocation function and the destructor (12.4, 12.5).

5.3.6 Alignof [expr.alignof]

1 An `alignof` expression yields the alignment requirement of its operand type. The operand shall be a *type-id* representing a complete effective object type or a reference to a complete effective object type.

2 The result is an integral constant of type `std::size_t`.

3 When `alignof` is applied to a reference type, the result shall be the alignment of the referenced type. When `alignof` is applied to an array type, the result shall be the alignment of the element type.

5.4 Explicit type conversion (cast notation) [expr.cast]

1 The result of the expression `(T)` *cast-expression* is of type `T`. The result is an lvalue if `T` is an lvalue reference type, otherwise the result is an rvalue. [*Note:* if `T` is a non-class type that is *cv-qualified*, the *cv-qualifiers* are ignored when determining the type of the resulting rvalue; see 3.10. — *end note*]

74) For non-zero-length arrays, this is the same as a pointer to the first element of the array created by that *new-expression*. Zero-length arrays do not have a first element.

75) This implies that an object cannot be deleted using a pointer of type `void*` because there are no objects of type `void`.

2 An explicit type conversion can be expressed using functional notation (5.2.3), a type conversion operator (`dynamic_cast`, `static_cast`, `reinterpret_cast`, `const_cast`), or the *cast* notation.

cast-expression:
unary-expression
`(` *type-id* `)` *cast-expression*

3 Any type conversion not mentioned below and not explicitly defined by the user (12.3) is ill-formed.

4 The conversions performed by

— a `const_cast` (5.2.11),

— a `static_cast` (5.2.9),

— a `static_cast` followed by a `const_cast`,

— a `reinterpret_cast` (5.2.10), or

— a `reinterpret_cast` followed by a `const_cast`,

can be performed using the cast notation of explicit type conversion. The same semantic restrictions and behaviors apply, with the exception that in performing a `static_cast` in the following situations the conversion is valid even if the base class is inaccessible:

— a pointer to an object of derived class type or an lvalue or rvalue of derived class type may be explicitly converted to a pointer or reference to an unambiguous base class type, respectively;

— a pointer to member of derived class type may be explicitly converted to a pointer to member of an unambiguous non-virtual base class type;

— a pointer to an object of an unambiguous non-virtual base class type, an lvalue or rvalue of an unambiguous non-virtual base class type, or a pointer to member of an unambiguous non-virtual base class type may be explicitly converted to a pointer, a reference, or a pointer to member of a derived class type, respectively.

If a conversion can be interpreted in more than one of the ways listed above, the interpretation that appears first in the list is used, even if a cast resulting from that interpretation is ill-formed. If a conversion can be interpreted in more than one way as a `static_cast` followed by a `const_cast`, the conversion is ill-formed. [*Example:*

```
struct A { };
struct I1 : A { };
struct I2 : A { };
struct D : I1, I2 { };
A *foo( D *p ) {
  return (A*)( p ); // ill-formed static_cast interpretation
}
```

— *end example*]

5 The operand of a cast using the cast notation can be an rvalue of type "pointer to incomplete class type". The destination type of a cast using the cast notation can be "pointer to incomplete class type". If both the operand and destination types are class types and one or both are incomplete, it is unspecified whether the `static_cast` or the `reinterpret_cast` interpretation is used, even if there is an inheritance relationship between the two classes. [*Note:* For example, if the classes were defined later in the translation unit, a multi-pass compiler would be permitted to interpret a cast between pointers to the classes as if the class types were complete at the point of the cast. — *end note*]

5.5 Pointer-to-member operators [expr.mptr.oper]

1 The pointer-to-member operators `->*` and `.*` group left-to-right.

pm-expression:
cast-expression
pm-expression `.*` *cast-expression*
pm-expression `->*` *cast-expression*

2 The binary operator `.*` binds its second operand, which shall be of type "pointer to member of `T`" (where `T` is a completely-defined effective class type) to its first operand, which shall be of class `T` or of a class of which `T` is an unambiguous and accessible base class. The result is an object or a function of the type specified by the second operand.

3 The binary operator `->*` binds its second operand, which shall be of type "pointer to member of `T`" (where `T` is a completely-defined effective class type) to its first operand, which shall be of type "pointer to `T`" or "pointer to a class of which `T` is an unambiguous and accessible base class." The result is an object or a function of the type specified by the second operand.

4 The first operand is called the *object expression*. If the dynamic type of the object expression does not contain the member to which the pointer refers, the behavior is undefined.

5 The restrictions on *cv*-qualification, and the manner in which the *cv*-qualifiers of the operands are combined to produce the *cv*-qualifiers of the result, are the same as the rules for `E1.E2` given in 5.2.5. [*Note:* it is not possible to use a pointer to member that refers to a `mutable` member to modify a `const` class object. For example,

```
struct S {
  S() : i(0) { }
  mutable int i;
};
void f()
{
const S cs;
int S::* pm = &S::i;            // pm refers to mutable member S::i
cs.*pm = 88;                    // ill-formed: cs is a const object
}
```

— *end note*]

6 If the result of `.*` or `->*` is a function, then that result can be used only as the operand for the function call operator `()`. [*Example:*

```
(ptr_to_obj->*ptr_to_mfct)(10);
```

calls the member function denoted by `ptr_to_mfct` for the object pointed to by `ptr_to_obj`. — *end example*] The result of a `.*` expression is an lvalue only if its first operand is an lvalue and its second operand is a pointer to data member. The result of an `->*` expression is an lvalue only if its second operand is a pointer to data member. If the second operand is the null pointer to member value (4.11), the behavior is undefined.

5.6 Multiplicative operators [expr.mul]

1 The multiplicative operators `*`, `/`, and `%` group left-to-right.

multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*

2 The operands of `*` and `/` shall have arithmetic or enumeration type; the operands of `%` shall have integral or enumeration type. The usual arithmetic conversions are performed on the operands and determine the type of the result.

3 The binary `*` operator indicates multiplication.

4 The binary `/` operator yields the quotient, and the binary `%` operator yields the remainder from the division of the first expression by the second. If the second operand of `/` or `%` is zero the behavior is undefined. For integeral operands the `/` operator yields the algebraic quotient with any fractional part discarded;[76] if the quotient `a/b` is representable in the type of the result, `(a/b)*b + a%b` is equal to `a`.

5.7 Additive operators [expr.add]

1 The additive operators `+` and `-` group left-to-right. The usual arithmetic conversions are performed for operands of arithmetic or enumeration type.

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

For addition, either both operands shall have arithmetic or enumeration type, or one operand shall be a pointer to a completely-defined effective object type and the other shall have integral or enumeration type.

2 For subtraction, one of the following shall hold:

— both operands have arithmetic or enumeration type; or

— both operands are pointers to cv-qualified or cv-unqualified versions of the same completely-defined effective object type; or

— the left operand is a pointer to a completely-defined effective object type and the right operand has integral or enumeration type.

3 The result of the binary `+` operator is the sum of the operands. The result of the binary `-` operator is the difference resulting from the subtraction of the second operand from the first.

4 For the purposes of these operators, a pointer to a nonarray object behaves the same as a pointer to the first element of an array of length one with the type of the object as its element type.

5 When an expression that has integral type is added to or subtracted from a pointer, the result has the type of the pointer operand. If the pointer operand points to an element of an array object, and the array is large enough, the result points to an element offset from the original element such that the difference of the subscripts of the resulting and original array elements equals the integral expression. In other words, if the expression `P` points to the i-th element of an array object, the expressions `(P)+N` (equivalently, `N+(P)`) and `(P)-N` (where `N` has the value n) point to, respectively, the $i + n$-th and $i - n$-th elements of the array object, provided they exist. Moreover, if the expression `P` points to the last element of an array object, the expression `(P)+1` points one past the last element of the array object, and if the expression `Q` points one past the last element of an array object, the expression `(Q)-1` points to the last element of the array

76) This is often called truncation towards zero.

object. If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

6 When two pointers to elements of the same array object are subtracted, the result is the difference of the subscripts of the two array elements. The type of the result is an implementation-defined signed integral type; this type shall be the same type that is defined as `std::ptrdiff_t` in the `<cstddef>` header (18.1). As with any other arithmetic overflow, if the result does not fit in the space provided, the behavior is undefined. In other words, if the expressions `P` and `Q` point to, respectively, the i-th and j-th elements of an array object, the expression `(P)-(Q)` has the value $i - j$ provided the value fits in an object of type `std::ptrdiff_t`. Moreover, if the expression `P` points either to an element of an array object or one past the last element of an array object, and the expression `Q` points to the last element of the same array object, the expression `((Q)+1)-(P)` has the same value as `((Q)-(P))+1` and as `-((P)-((Q)+1))`, and has the value zero if the expression `P` points one past the last element of the array object, even though the expression `(Q)+1` does not point to an element of the array object. Unless both pointers point to elements of the same array object, or one past the last element of the array object, the behavior is undefined.[77]

7 If the value 0 is added to or subtracted from a pointer value, the result compares equal to the original pointer value. If two pointers point to the same object or both point one past the end of the same array or both are null, and the two pointers are subtracted, the result compares equal to the value 0 converted to the type `std::ptrdiff_t`.

5.8 Shift operators [expr.shift]

1 The shift operators `<<` and `>>` group left-to-right.

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

The operands shall be of integral or enumeration type and integral promotions are performed. The type of the result is that of the promoted left operand. The behavior is undefined if the right operand is negative, or greater than or equal to the length in bits of the promoted left operand.

2 The value of `E1 <<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bit positions; vacated bits are zero-filled. If `E1` has an unsigned type, the value of the result is `E1` multiplied by the quantity 2 raised to the power `E2`, reduced modulo `ULLONG_MAX+1` if `E1` has type `unsigned long long int`, `ULONG_MAX+1` if `E1` has type unsigned long `int`, `UINT_MAX+1` otherwise. [*Note:* the constants `ULLONG_MAX`, `ULONG_MAX`, and `UINT_MAX` are defined in the header `<climits>`. — *end note*]

3 The value of `E1 >>E2` is `E1` right-shifted `E2` bit positions. If `E1` has an unsigned type or if `E1` has a signed type and a nonnegative value, the value of the result is the integral part of the quotient of `E1` divided by

77) Another way to approach pointer arithmetic is first to convert the pointer(s) to character pointer(s): In this scheme the integral value of the expression added to or subtracted from the converted pointer is first multiplied by the size of the object originally pointed to, and the resulting pointer is converted back to the original type. For pointer subtraction, the result of the difference between the character pointers is similarly divided by the size of the object originally pointed to.

When viewed in this way, an implementation need only provide one extra byte (which might overlap another object in the program) just after the end of the object in order to satisfy the "one past the last element" requirements.

the quantity 2 raised to the power `E2`. If `E1` has a signed type and a negative value, the resulting value is implementation-defined.

5.9 Relational operators [expr.rel]

1 The relational operators group left-to-right. [*Example:* `a<b<c` means `(a<b)<c` and *not* `(a<b)&&(b<c)`. — *end example*]

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

The operands shall have arithmetic, enumeration, or pointer type, or type `std::nullptr_t`. The operators `<` (less than), `>` (greater than), `<=` (less than or equal to), and `>=` (greater than or equal to) all yield `false` or `true`. The type of the result is `bool`.

2 The usual arithmetic conversions are performed on operands of arithmetic or enumeration type. Pointer conversions (4.10) and qualification conversions (4.4) are performed on pointer operands (or on a pointer operand and a null pointer constant) to bring them to their *composite pointer type*. If one operand is a null pointer constant, the composite pointer type is the type of the other operand. Otherwise, if one of the operands has type "pointer to *cv1* `void`," then the other has type "pointer to *cv2 T*" and the composite pointer type is "pointer to *cv12* `void`," where *cv12* is the union of *cv1* and *cv2*. Otherwise, the composite pointer type is a pointer type similar (4.4) to the type of one of the operands, with a cv-qualification signature (4.4) that is the union of the cv-qualification signatures of the operand types. [*Note:* this implies that any pointer can be compared to a null pointer constant and that any object pointer can be compared to a pointer to (possibly cv-qualified) `void`. — *end note*] [*Example:*

```
void *p;
const int *q;
int **pi;
const int *const *pci;
void ct() {
  p <= q;           // Both converted to const void* before comparison
  pi <= pci;        // Both converted to const int *const * before comparison
}
```

— *end example*] Pointers to objects or functions of the same type (after pointer conversions) can be compared, with a result defined as follows:

— If two pointers `p` and `q` of the same type point to the same object or function, or both point one past the end of the same array, or are both null, then `p<=q` and `p>=q` both yield `true` and `p<q` and `p>q` both yield `false`.

— If two pointers `p` and `q` of the same type point to different objects that are not members of the same object or elements of the same array or to different functions, or if only one of them is null, the results of `p<q`, `p>q`, `p<=q`, and `p>=q` are unspecified.

— If two pointers point to non-static data members of the same object, or to subobjects or array elements of such members, recursively, the pointer to the later declared member compares greater provided the two members have the same access control (Clause 11) and provided their class is not a union.

— If two pointers point to non-static data members of the same object with different access control (Clause 11) the result is unspecified.

— If two pointers point to data members of the same union object, they compare equal (after conversion to `void*`, if necessary). If two pointers point to elements of the same array or one beyond the end of the array, the pointer to the object with the higher subscript compares higher.

— Other pointer comparisons are unspecified.

3 If two operands of type `std::nullptr_t` are compared, the result is `true` if the operator is `<=` or `>=`, and `false` otherwise.

4 If both operands (after conversions) are of arithmetic type, each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

5.10 Equality operators [expr.eq]

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

1 The `==` (equal to) and the `!=` (not equal to) operators have the same semantic restrictions, conversions, and result type as the relational operators except for their lower precedence and truth-value result. [*Note:* `a<b == c<d` is `true` whenever `a<b` and `c<d` have the same truth-value. — *end note*] Pointers to objects or functions of the same type (after pointer conversions) can be compared for equality. Two pointers of the same type compare equal if and only if they are both null, both point to the same function, or both represent the same address (3.9.2).

2 In addition, pointers to members can be compared, or a pointer to member and a null pointer constant. Pointer to member conversions (4.11) and qualification conversions (4.4) are performed to bring them to a common type. If one operand is a null pointer constant, the common type is the type of the other operand. Otherwise, the common type is a pointer to member type similar (4.4) to the type of one of the operands, with a cv-qualification signature (4.4) that is the union of the cv-qualification signatures of the operand types. [*Note:* this implies that any pointer to member can be compared to a null pointer constant. — *end note*] If both operands are null, they compare equal. Otherwise if only one is null, they compare unequal. Otherwise if either is a pointer to a virtual member function, the result is unspecified. Otherwise they compare equal if and only if they would refer to the same member of the same most derived object (1.8) or the same subobject if they were dereferenced with a hypothetical object of the associated class type. [*Example:*

```
struct B {
  int f();
};
struct L : B { };
struct R : B { };
struct D : L, R { };

int (B::*pb)() = &B::f;
int (L::*pl)() = pb;
int (R::*pr)() = pb;
int (D::*pdl)() = pl;
int (D::*pdr)() = pr;
bool x = (pdl == pdr);          // false
```

— *end example*]

3 If two operands of type `std::nullptr_t` are compared, the result is `true` if the operator is `==`, and `false` otherwise.

4 Each of the operators shall yield `true` if the specified relationship is true and `false` if it is false.

5.11 Bitwise AND operator [expr.bit.and]

and-expression:
equality-expression
and-expression `&` *equality-expression*

1 The usual arithmetic conversions are performed; the result is the bitwise AND function of the operands. The operator applies only to integral or enumeration operands.

5.12 Bitwise exclusive OR operator [expr.xor]

exclusive-or-expression:
and-expression
exclusive-or-expression `^` *and-expression*

1 The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral or enumeration operands.

5.13 Bitwise inclusive OR operator [expr.or]

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression `|` *exclusive-or-expression*

1 The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral or enumeration operands.

5.14 Logical AND operator [expr.log.and]

logical-and-expression:
inclusive-or-expression
logical-and-expression `&&` *inclusive-or-expression*

1 The `&&` operator groups left-to-right. The operands are both contextually converted to type `bool` (Clause 4). The result is `true` if both operands are `true` and `false` otherwise. Unlike `&`, `&&` guarantees left-to-right evaluation: the second operand is not evaluated if the first operand is `false`.

2 The result is a `bool`. If the second expression is evaluated, every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second expression.

5.15 Logical OR operator [expr.log.or]

logical-or-expression:
logical-and-expression
logical-or-expression || *logical-and-expression*

1 The || operator groups left-to-right. The operands are both contextually converted to `bool` (Clause 4). It returns `true` if either of its operands is `true`, and `false` otherwise. Unlike |, || guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the first operand evaluates to `true`.

2 The result is a `bool`. All side effects of the first expression except for destruction of temporaries (12.2) happen before the second expression is evaluated.

5.16 Conditional operator [expr.cond]

conditional-expression:
logical-or-expression
logical-or-expression `?` *expression* `:` *assignment-expression*

1 Conditional expressions group right-to-left. The first expression is contextually converted to `bool` (Clause 4). It is evaluated and if it is `true`, the result of the conditional expression is the value of the second expression, otherwise that of the third expression. Only one of the second and third expressions is evaluated. Every value computation and side effect associated with the first expression is sequenced before every value computation and side effect associated with the second or third expression.

2 If either the second or the third operand has type (possibly cv-qualified) `void`, then the lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the second and third operands, and one of the following shall hold:

— The second or the third operand (but not both) is a *throw-expression* (15.1); the result is of the type of the other and is an rvalue.

— Both the second and the third operands have type `void`; the result is of type `void` and is an rvalue. [*Note:* this includes the case where both operands are *throw-expressions*. — *end note*]

3 Otherwise, if the second and third operand have different types, and either has (possibly cv-qualified) class type, an attempt is made to convert each of those operands to the type of the other. The process for determining whether an operand expression `E1` of type `T1` can be converted to match an operand expression `E2` of type `T2` is defined as follows:

— If `E2` is an lvalue: `E1` can be converted to match `E2` if `E1` can be implicitly converted (Clause 4) to the type "lvalue reference to `T2`", subject to the constraint that in the conversion the reference must bind directly (8.5.3) to `E1`.

— If `E2` is an rvalue, or if the conversion above cannot be done:

— if `E1` and `E2` have class type, and the underlying class types are the same or one is a base class of the other: `E1` can be converted to match `E2` if the class of `T2` is the same type as, or a base class of, the class of `T1`, and the cv-qualification of `T2` is the same cv-qualification as, or a greater cv-qualification than, the cv-qualification of `T1`. If the conversion is applied, `E1` is changed to an rvalue of type `T2` by copy-initializing a temporary of type `T2` from `E1` and using that temporary as the converted operand.

— Otherwise (i.e., if `E1` or `E2` has a nonclass type, or if they both have class types but the underlying classes are not either the same or one a base class of the other): `E1` can be converted to match `E2` if `E1` can be implicitly converted to the type that expression `E2` would have if `E2` were converted to an rvalue (or the type it has, if `E2` is an rvalue).

Using this process, it is determined whether the second operand can be converted to match the third operand, and whether the third operand can be converted to match the second operand. If both can be converted, or one can be converted but the conversion is ambiguous, the program is ill-formed. If neither can be converted, the operands are left unchanged and further checking is performed as described below. If exactly one conversion is possible, that conversion is applied to the chosen operand and the converted operand is used in place of the original operand for the remainder of this section.

4 If the second and third operands are lvalues and have the same type, the result is of that type and is an lvalue and it is a bit-field if the second or the third operand is a bit-field, or if both are bit-fields.

5 Otherwise, the result is an rvalue. If the second and third operands do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is used to determine the conversions (if any) to be applied to the operands (13.3.1.2, 13.6). If the overload resolution fails, the program is ill-formed. Otherwise, the conversions thus determined are applied, and the converted operands are used in place of the original operands for the remainder of this section.

6 Lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are performed on the second and third operands. After those conversions, one of the following shall hold:

— The second and third operands have the same type; the result is of that type. If the operands have class type, the result is an rvalue temporary of the result type, which is copy-initialized from either the second operand or the third operand depending on the value of the first operand.

— The second and third operands have arithmetic or enumeration type; the usual arithmetic conversions are performed to bring them to a common type, and the result is of that type.

— The second and third operands have pointer type, or one has pointer type and the other is a null pointer constant; pointer conversions (4.10) and qualification conversions (4.4) are performed to bring them to their composite pointer type (5.9). The result is of the composite pointer type.

— The second and third operands have pointer to member type, or one has pointer to member type and the other is a null pointer constant; pointer to member conversions (4.11) and qualification conversions (4.4) are performed to bring them to a common type, whose cv-qualification shall match the cv-qualification of either the second or the third operand. The result is of the common type.

5.17 Assignment and compound assignment operators [expr.ass]

1 The assignment operator (`=`) and the compound assignment operators all group right-to-left. All require a modifiable lvalue as their left operand and return an lvalue referring to the left operand. The result in all cases is a bit-field if the left operand is a bit-field. In all cases, the assignment is sequenced after the value computation of the right and left operands, and before the value computation of the assignment expression. With respect to an indeterminately-sequenced function call, the operation of a compound assignment is a single evaluation. [*Note:* Therefore, a function call shall not intervene between the lvalue-to-rvalue conversion and the side effect associated with any single compound assignment operator. — *end note*]

assignment-expression:
conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression

assignment-operator: one of
`= *= /= %= += -= >>= <<= &= ^= |=`

2 In simple assignment (`=`), the value of the expression replaces that of the object referred to by the left operand.

3 If the left operand is not of class type, the expression is implicitly converted (Clause 4) to the cv-unqualified type of the left operand.

4 If the left operand is of class type, the class shall be complete. Assignment to objects of a class is defined by the copy assignment operator (12.8, 13.5.3).

5 [*Note:* For class objects, assignment is not in general the same as initialization (8.5, 12.1, 12.6, 12.8). — *end note*]

6 When the left operand of an assignment operator denotes a reference to `T`, the operation assigns to the object of type `T` denoted by the reference.

7 The behavior of an expression of the form `E1` *op* `= E2` is equivalent to `E1 = E1` *op* `E2` except that `E1` is evaluated only once. In `+=` and `-=`, `E1` shall either have arithmetic type or be a pointer to a possibly cv-qualified completely-defined effective object type. In all other cases, `E1` shall have arithmetic type.

8 If the value being stored in an object is accessed from another object that overlaps in any way the storage of the first object, then the overlap shall be exact and the two objects shall have the same type, otherwise the behavior is undefined.

9 A *braced-init-list* may appear on the right-hand side of

— an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T(v)` except that no narrowing conversion (8.5.4) is allowed. The meaning of `x={}` is `x=T()`.

— an assignment defined by a user-defined assignment operator, in which case the initializer list is passed as the argument to the operator function.

[*Example:*

```
complex<double> z;
z = { 1,2 };              // meaning z.operator=(1,2)
z += { 1, 2 };            // meaning z.operator+=(1,2)
a = b = { 1 };            // meaning a=b=1;
a = { 1 } = b;            // syntax error
```

— *end example*]

5.18 Comma operator [expr.comma]

1 The comma operator groups left-to-right.

expression:
assignment-expression
expression , *assignment-expression*

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the left expression. Every value computation and side effect associated with the left expression is sequenced before every value computation and side effect associated with the right expression. The type and value of the result are the type and value of the right operand; the result is an lvalue if its right operand is an lvalue, and is a bit-field if its right operand is an lvalue and a bit-field.

2 In contexts where comma is given a special meaning, [*Example:* in lists of arguments to functions (5.2.2) and lists of initializers (8.5) — *end example*] the comma operator as described in Clause 5 can appear only in parentheses. [*Example:*

```
f(a, (t=3, t+2), c);
```

has three arguments, the second of which has the value `5`. — *end example*]

5.19 Constant expressions [expr.const]

1 Certain contexts require expressions that satisfy additional requirements as detailed in this sub-clause. Such expressions are called *constant expressions*. [*Note:* Those expressions can be evaluated during translation. — *end note*]

constant-expression:
conditional-expression

2 A *conditional-expression* is a constant expression unless it involves one of the following as a potentially evaluated subexpression (3.2), but subexpressions of logical AND (5.14), logical OR (5.15), and conditional (5.16) operations that are not evaluated are not considered [*Note:* an overloaded operator invokes a function. — *end note*]:

- `this` (5.1) unless it appears as the *postfix-expression* in a class member access expression, including the result of the implicit transformation in the body of a non-static member function (9.3.1);
- an invocation of a function other than a constexpr function or a constexpr constructor [*Note:* overload resolution (13.3) is applied as usual — *end note*];
- a *lambda-expression* (5.1.1);
- an lvalue-to-rvalue conversion (4.1) unless it is applied to
  - an lvalue of effective integral type that refers to a non-volatile const variable or static data member initialized with constant expressions, or
  - an lvalue of effective literal type that refers to a non-volatile object defined with `constexpr`, or that refers to a sub-object of such an object;
- an array-to-pointer conversion (4.2) that is applied to an lvalue that designates an object with thread or automatic storage duration;
- a unary operator `&` (5.3.1) that is applied to an lvalue that designates an object with thread or automatic storage duration;
- an *id-expression* that refers to a variable or data member of reference type;
- a type conversion from a floating-point type to an integral type (4.9) unless the conversion is directly applied to a floating-point literal;
- a dynamic cast (5.2.7);
- a type conversion from a pointer or pointer-to-member type to a literal type [*Note:* a user-defined conversion invokes a function — *end note*];
- a pseudo-destructor call (5.2.4);
- a class member access (5.2.5) unless its *postfix-expression* is of effective literal type or of pointer to effective literal type;
- increment or decrement operations (5.2.6, 5.3.2);
- a typeid expression (5.2.8) whose operand is of a polymorphic class type;
- a *new-expression* (5.3.4);
- a *delete-expression* (5.3.5);
- a subtraction (5.7) where both operands are pointers;
- a relational (5.9) or equality (5.10) operator where at least one of the operands is a pointer;
- an assignment or a compound assignment (5.17); or
- a *throw-expression* (15.1).

3 A constant expression is an *integral constant expression* if it is of integral or enumeration type or if it is an archetype that has the concept requirement `std::IntegralConstantExpressionType<T>` (14.9.4).

[*Note:* such expressions may be used as array bounds (8.3.4, 5.3.4), as case expressions (6.4.2), as bit-field lengths (9.6), as enumerator initializers (7.2), as static member initializers (9.4.2), and as integral or enumeration non-type template arguments (14.3). — *end note*]

4 If an expression of effective literal class type is used in a context where an integral constant expression is required, then that class type shall have a single non-explicit conversion function to an integral or enumeration type and that conversion function shall be constexpr. [*Example:*

```
struct A {
  constexpr A(int i) : val(i) { }
  constexpr operator int() { return val; }
  constexpr operator long() { return 43; }
private:
  int val;
};
template<int> struct X { };
constexpr A a = 42;
X<a> x;             // OK: unique conversion to int
int ary[a];         // error: ambiguous conversion
```

— *end example*]

5 An expression is a *potential constant expression* if it is a constant expression when all occurrences of function parameters are replaced by arbitrary constant expressions of the appropriate type.

6 Statements [stmt.stmt]

1 Except as indicated, statements are executed in sequence.

statement:
labeled-statement
attribute-specifier$_{opt}$ *expression-statement*
attribute-specifier$_{opt}$ *compound-statement*
attribute-specifier$_{opt}$ *selection-statement*
attribute-specifier$_{opt}$ *iteration-statement*
attribute-specifier$_{opt}$ *jump-statement*
declaration-statement
attribute-specifier$_{opt}$ *try-block*
late-checked-block

The optional *attribute-specifier* appertains to the respective statement.

6.1 Labeled statement [stmt.label]

1 A statement can be labeled.

labeled-statement:
attribute-specifier$_{opt}$ *identifier* `:` *statement*
attribute-specifier$_{opt}$ `case` *constant-expression* `:` *statement*
attribute-specifier$_{opt}$ `default` `:` *statement*

The optional *attribute-specifier* appertains to the label. An identifier label declares the identifier. The only use of an identifier label is as the target of a `goto`. The scope of a label is the function in which it appears. Labels shall not be redeclared within a function. A label can be used in a `goto` statement before its definition. Labels have their own name space and do not interfere with other identifiers.

2 Case labels and default labels shall occur only in switch statements.

6.2 Expression statement [stmt.expr]

1 Expression statements have the form

expression-statement:
expression$_{opt}$ `;`

The expression is evaluated and its value is discarded. The lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not applied to the expression. All side effects from an expression statement are completed before the next statement is executed. An expression statement with the expression missing is called a null statement. [*Note:* Most statements are expression statements — usually assignments or function calls. A null statement is useful to carry a label just before the `}` of a compound statement and to supply a null body to an iteration statement such as a `while` statement (6.5.1). — *end note*]

6.3 Compound statement or block [stmt.block]

1 So that several statements can be used where one is expected, the compound statement (also, and equivalently, called "block") is provided.

compound-statement:
`{` *statement-seq*$_{opt}$ `}`

statement-seq:
statement
statement-seq statement

A compound statement defines a local scope (3.3). [*Note:* a declaration is a *statement* (6.7). — *end note*]

6.4 Selection statements [stmt.select]

1 Selection statements choose one of several flows of control.

selection-statement:
`if (` *condition* `)` *statement*
`if (` *condition* `)` *statement* `else` *statement*
`switch (` *condition* `)` *statement*

condition:
expression
type-specifier-seq attribute-specifier$_{opt}$ *declarator* `=` *initializer-clause*
type-specifier-seq attribute-specifier$_{opt}$ *declarator braced-init-list*

See 8.3 for the optional *attribute-specifier* in a condition. In Clause 6, the term *substatement* refers to the contained *statement* or *statement*s that appear in the syntax notation. The substatement in a *selection-statement* (each substatement, in the `else` form of the `if` statement) implicitly defines a local scope (3.3). If the substatement in a selection-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original substatement. [*Example:*

```
if (x)
  int i;
```

can be equivalently rewritten as

```
if (x) {
  int i;
}
```

Thus after the `if` statement, `i` is no longer in scope. — *end example*]

2 The rules for *condition*s apply both to *selection-statement*s and to the `for` and `while` statements (6.5). The *declarator* shall not specify a function or an array. If the `auto` *type-specifier* appears in the *type-specifier-seq*, the type of the identifier being declared is deduced from the initializer as described in 7.1.6.4.

3 A name introduced by a declaration in a *condition* (either introduced by the *type-specifier-seq* or the *declarator* of the condition) is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is re-declared in the outermost block of a substatement controlled by the condition, the declaration that re-declares the name is ill-formed. [*Example:*

```
if (int x = f()) {
  int x;            // ill-formed, redeclaration of x
}
else {
  int x;            // ill-formed, redeclaration of x
}
```

— *end example*]

4 The value of a *condition* that is an initialized declaration in a statement other than a `switch` statement is the value of the declared variable contextually converted to `bool` (Clause 4). If that conversion is ill-formed, the program is ill-formed. The value of a *condition* that is an initialized declaration in a `switch` statement is the value of the declared variable if it has integral or enumeration type, or of that variable implicitly converted

to integral or enumeration type otherwise. The value of a *condition* that is an expression is the value of the expression, contextually converted to `bool` for statements other than `switch`; if that conversion is ill-formed, the program is ill-formed. The value of the condition will be referred to as simply "the condition" where the usage is unambiguous.

5 If a *condition* can be syntactically resolved as either an expression or the declaration of a local name, it is interpreted as a declaration.

6.4.1 The `if` statement [stmt.if]

1 If the condition (6.4) yields `true` the first substatement is executed. If the `else` part of the selection statement is present and the condition yields `false`, the second substatement is executed. In the second form of `if` statement (the one including `else`), if the first substatement is also an `if` statement then that inner `if` statement shall contain an `else` part.[78]

6.4.2 The `switch` statement [stmt.switch]

1 The `switch` statement causes control to be transferred to one of several statements depending on the value of a condition.

2 The condition shall be of integral type, enumeration type, or of a class type for which a single non-explicit conversion function to integral or enumeration type exists (12.3). If the condition is of class type, the condition is converted by calling that conversion function, and the result of the conversion is used in place of the original condition for the remainder of this section. Integral promotions are performed. Any statement within the `switch` statement can be labeled with one or more case labels as follows:

`case` *constant-expression* `:`

where the *constant-expression* shall be an integral constant expression (5.19). The integral constant expression is implicitly converted to the promoted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion to the promoted type of the switch condition.

3 There shall be at most one label of the form

```
default :
```

within a `switch` statement.

4 Switch statements can be nested; a `case` or `default` label is associated with the smallest switch enclosing it.

5 When the `switch` statement is executed, its condition is evaluated and compared with each case constant. If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition, and if there is a `default` label, control passes to the statement labeled by the default label. If no case matches and if there is no `default` then none of the statements in the switch is executed.

6 `case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To exit from a switch, see `break`, 6.6.1. [*Note:* usually, the substatement that is the subject of a switch is compound and `case` and `default` labels appear on the top-level statements contained within the (compound) substatement, but this is not required. Declarations can appear in the substatement of a *switch-statement*. — *end note*]

6.5 Iteration statements [stmt.iter]

1 Iteration statements specify looping.

78) In other words, the `else` is associated with the nearest un-elsed `if`.

iteration-statement:
`while` `(` *condition* `)` *statement*
`do` *statement* `while` `(` *expression* `)` `;`
`for` `(` *for-init-statement* *condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`for` `(` *for-range-declaration* `:` *expression* `)` *statement*

for-init-statement:
expression-statement
simple-declaration

for-range-declaration:
type-specifier-seq *attribute-specifier*$_{opt}$ *declarator*

[*Note:* a *for-init-statement* ends with a semicolon. — *end note*]

2 The substatement in an *iteration-statement* implicitly defines a local scope (3.3) which is entered and exited each time through the loop.

If the substatement in an iteration-statement is a single statement and not a *compound-statement,* it is as if it was rewritten to be a compound-statement containing the original statement. [*Example:*

```
while (--x >= 0)
  int i;
```

can be equivalently rewritten as

```
while (--x >= 0) {
  int i;
}
```

3 Thus after the `while` statement, `i` is no longer in scope. — *end example*]

4 [*Note:* The requirements on *condition*s in iteration statements are described in 6.4. — *end note*]

5 A loop that, outside of the *for-init-statement* in the case of a `for` statement,

— performs no I/O operations, and

— does not access or modify volatile objects, and

— performs no synchronization or atomic operations

may be assumed by the implementation to terminate. [*Note:* This is intended to allow compiler transformations, such as removal of empty loops, even when termination cannot be proven. — *end note*]

6.5.1 The `while` statement [stmt.while]

1 In the `while` statement the substatement is executed repeatedly until the value of the condition (6.4) becomes `false`. The test takes place before each execution of the substatement.

2 When the condition of a while statement is a declaration, the scope of the variable that is declared extends from its point of declaration (3.3.1) to the end of the while *statement*. A while statement of the form

```
while (T t = x) statement
```

is equivalent to

```
label:
{                       // start of condition scope
  T t = x;
  if (t) {
```

```
    statement
    goto label;
  }
}                    // end of condition scope
```

The object created in a condition is destroyed and created with each iteration of the loop. [*Example:*

```
struct A {
  int val;
  A(int i) : val(i) { }
  ~A() { }
  operator bool() { return val != 0; }
};
int i = 1;
while (A a = i) {
  // ...
  i = 0;
}
```

In the while-loop, the constructor and destructor are each called twice, once for the condition that succeeds and once for the condition that fails. — *end example*]

6.5.2 The `do` statement [stmt.do]

1 The expression is contextually converted to `bool` (Clause 4); if that conversion is ill-formed, the program is ill-formed.

2 In the `do` statement the substatement is executed repeatedly until the value of the expression becomes `false`. The test takes place after each execution of the statement.

6.5.3 The `for` statement [stmt.for]

1 The `for` statement

`for (` *for-init-statement* *condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

is equivalent to

```
{
    for-init-statement
    while ( condition ) {
        statement
        expression ;
    }
}
```

except that names declared in the *for-init-statement* are in the same declarative-region as those declared in the *condition*, and except that a `continue` in *statement* (not enclosed in another iteration statement) will execute *expression* before re-evaluating *condition*. [*Note:* Thus the first statement specifies initialization for the loop; the condition (6.4) specifies a test, made before each iteration, such that the loop is exited when the condition becomes `false`; the expression often specifies incrementing that is done after each iteration. — *end note*]

2 Either or both of the condition and the expression can be omitted. A missing *condition* makes the implied `while` Clause equivalent to `while(true)`.

3 If the *for-init-statement* is a declaration, the scope of the name(s) declared extends to the end of the *for-statement*. [*Example:*

```
int i = 42;
int a[10];

for (int i = 0; i < 10; i++)
  a[i] = i;

int j = i;          // j = 42
```

— *end example*]

6.5.4 The range-based `for` statement [stmt.ranged]

1 The range-base `for` statement

`for` (*for-range-declaration* : *expression*) *statement*

is equivalent to

```
{
  auto && __range = ( expression );
  for ( auto __begin = std::Range<_RangeT>::begin(__range),
             __end = std::Range<_RangeT>::end(__range);
        __begin != __end;
        ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

where `__range`, `__begin`, and `__end` are variables defined for exposition only, and `_RangeT` is the type of the *expression*.

[*Example:*

```
int array[5] = { 1, 2, 3, 4, 5 };
for (int& x : array)
  x *= 2;
```

— *end example*]

2 If the header `<iterator_concepts>` (24.1) is not included prior to a use of the range-based `for` statement, the program is ill-formed.

6.6 Jump statements [stmt.jump]

1 Jump statements unconditionally transfer control.

jump-statement:
`break` ;
`continue` ;
`return` *expression*$_{opt}$;
`return` *braced-init-list* ;
`goto` *identifier* ;

2 On exit from a scope (however accomplished), variables with automatic storage duration (3.7.3) that have been constructed in that scope are destroyed in the reverse order of their construction. [*Note:* For temporaries, see 12.2. — *end note*] Transfer out of a loop, out of a block, or back past an initialized variable with automatic storage duration involves the destruction of variables with automatic storage duration that

are in scope at the point transferred from but not at the point transferred to. (See 6.7 for transfers into blocks). [*Note:* However, the program can be terminated (by calling `std::exit()` or `std::abort()` (18.4), for example) without destroying class objects with automatic storage duration. — *end note*]

6.6.1 The `break` statement [stmt.break]

1 The `break` statement shall occur only in an *iteration-statement* or a `switch` statement and causes termination of the smallest enclosing *iteration-statement* or `switch` statement; control passes to the statement following the terminated statement, if any.

6.6.2 The `continue` statement [stmt.cont]

1 The `continue` statement shall occur only in an *iteration-statement* and causes control to pass to the loop-continuation portion of the smallest enclosing *iteration-statement*, that is, to the end of the loop. More precisely, in each of the statements

```
while (foo) {
  {
    // ...
  }
contin: ;
}
```

```
do {
  {
    // ...
  }
contin: ;
} while (foo);
```

```
for (;;) {
  {
    // ...
  }
contin: ;
}
```

a `continue` not contained in an enclosed iteration statement is equivalent to `goto contin`.

6.6.3 The `return` statement [stmt.return]

1 A function returns to its caller by the `return` statement.

2 A return statement without an expression can be used only in functions that do not return a value, that is, a function with the return type `void`, a constructor (12.1), or a destructor (12.4). A return statement with an expression of non-void type can be used only in functions returning a value; the value of the expression is returned to the caller of the function. The expression is implicitly converted to the return type of the function in which it appears. A return statement can involve the construction and copy of a temporary object (12.2). [*Note:* A copy operation associated with a return statement may be elided or considered as an rvalue for the purpose of overload resolution in selecting a constructor (12.8). — *end note*] A return statement with a *braced-init-list* initializes the object or reference to be returned from the function by copy-list-initialization (8.5.4) from the specified initializer list. [*Example:*

```
std::pair<std::string,int> f(const char* p, int x) {
  return {p,x};
}
```

— *end example*]

Flowing off the end of a function is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function.

3 A return statement with an expression of type "*cv* `void`" can be used only in functions with a return type of *cv* void; the expression is evaluated just before the function returns to its caller.

6.6.4 The `goto` statement [stmt.goto]

1 The `goto` statement unconditionally transfers control to the statement labeled by the identifier. The identifier

shall be a label (6.1) located in the current function.

6.7 Declaration statement [stmt.dcl]

1 A declaration statement introduces one or more new identifiers into a block; it has the form

declaration-statement:
block-declaration

If an identifier introduced by a declaration was previously declared in an outer block, the outer declaration is hidden for the remainder of the block, after which it resumes its force.

2 Variables with automatic storage duration (3.7.3) are initialized each time their *declaration-statement* is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).

3 It is possible to transfer into a block, but not in a way that bypasses declarations with initialization. A program that jumps[79] from a point where a local variable with automatic storage duration is not in scope to a point where it is in scope is ill-formed unless the variable has scalar type, class type with a trivial default constructor and a trivial destructor, a cv-qualified version of one of these types, or an array of one of the preceding types and is declared without an *initializer* (8.5). [*Example:*

```
void f() {
  // ...
  goto lx;          // ill-formed: jump into scope of a
  // ...
ly:
  X a = 1;
  // ...
lx:
  goto ly;          // OK, jump implies destructor
                    // call for a followed by construction
                    // again immediately following label ly
}
```

— *end example*]

4 The zero-initialization (8.5) of all local objects with static storage duration (3.7.1) or thread storage duration (3.7.2) is performed before any other initialization takes place. Constant initialization (3.6.2) of a local entity with static storage duration, if applicable, is performed before its block is first entered. An implementation is permitted to perform early initialization of other local objects with static or thread storage duration under the same conditions that an implementation is permitted to statically initialize an object with static or thread storage duration in namespace scope (3.6.2). Otherwise such an object is initialized the first time control passes through its declaration; such an object is considered initialized upon the completion of its initialization. If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the object is being initialized, the concurrent execution shall wait for completion of the initialization.[80] If control re-enters the declaration recursively while the object is being initialized, the behavior is undefined. [*Example:*

```
int foo(int i) {
  static int s = foo(2*i);      // recursive call - undefined
  return i+1;
}
```

79) The transfer from the condition of a `switch` statement to a `case` label is considered a jump in this respect.
80) The implementation must not introduce any deadlock around execution of the initializer.

— *end example*]

5 The destructor for a local object with static or thread storage duration will be executed if and only if the variable was constructed. [*Note:* 3.6.3 describes the order in which local objects with static and thread storage duration are destroyed. — *end note*]

6.8 Ambiguity resolution [stmt.ambig]

1 There is an ambiguity in the grammar involving *expression-statement*s and *declaration*s: An *expression-statement* with a function-style explicit type conversion (5.2.3) as its leftmost subexpression can be indistinguishable from a *declaration* where the first *declarator* starts with a `(`. In those cases the *statement* is a *declaration*. [*Note:* To disambiguate, the whole *statement* might have to be examined to determine if it is an *expression-statement* or a *declaration*. This disambiguates many examples. [*Example:* assuming `T` is a *simple-type-specifier* (7.1.6),

```
T(a)->m = 7;          // expression-statement
T(a)++;               // expression-statement
T(a,5)<<c;            // expression-statement

T(*d)(int);           // declaration
T(e)[5];              // declaration
T(f) = { 1, 2 };      // declaration
T(*g)(double(3));     // declaration
```

In the last example above, `g`, which is a pointer to `T`, is initialized to `double(3)`. This is of course ill-formed for semantic reasons, but that does not affect the syntactic analysis. — *end example*]

2 The remaining cases are *declaration*s. [*Example:*

```
class T {
  // ...
public:
  T();
  T(int);
  T(int, int);
};
T(a);                 // declaration
T(*b)();              // declaration
T(c)=7;               // declaration
T(d),e,f=3;           // declaration
extern int h;
T(g)(h,2);            // declaration
```

— *end example*] — *end note*]

3 The disambiguation is purely syntactic; that is, the meaning of the names occurring in such a statement, beyond whether they are *type-name*s or not, is not generally used in or changed by the disambiguation. Class templates are instantiated as necessary to determine if a qualified name is a *type-name*. Disambiguation precedes parsing, and a statement disambiguated as a declaration may be an ill-formed declaration. If, during parsing, a name in a template parameter is bound differently than it would be bound during a trial parse, the program is ill-formed. No diagnostic is required. [*Note:* This can occur only when the name is declared earlier in the declaration. — *end note*] [*Example:*

```
struct T1 {
  T1 operator()(int x) { return T1(x); }
  int operator=(int x) { return x; }
```

```
  T1(int) { }
};
struct T2 { T2(int){ } };
int a, (*(*b)(T2))(int), c, d;

void f() {
  // disambiguation requires this to be parsed as a declaration:
  T1(a) = 3,
  T2(4),                          // T2 will be declared as
  (*(*b)(T2(c)))(int(d));         // a variable of type T1
                                  // but this will not allow
                                  // the last part of the
                                  // declaration to parse
                                  // properly since it depends
                                  // on T2 being a type-name
}
```

— *end example*]

6.9 Late-checked block [stmt.late]

1 In a constrained context (14.10), a *late-checked block* treats the enclosed statements as if they were in an unconstrained context. Outside of a constrained context, the late-checked block has no effect. [*Note:* in a late-checked block, template parameters do not behave as if they were replaced with their corresponding archetypes. Thus, template parameters imply the existence of dependent types, type-dependent expressions, and dependent names as in an unconstrained template. Furthermore, names at requirements scope (3.3.8) are not visible. — *end note*]

late-checked-block:
`late_check` *compound-statement*

2 [*Example:*

```
concept Semigroup<typename T> {
  T::T(const T&);
  T operator+(T, T);
}

concept_map Semigroup<int> {
  int operator+(int x, int y) { return x * y; }
}

template<Semigroup T>
T add(T x, T y) {
  T r = x + y;       // uses Semigroup<T>::operator+
  late_check {
    r = x + y;       // uses operator+ found at instantiation time (not considering Semigroup<T>::operator+)
  }
  return r;
}
```

— *end example*]

3 [*Note:* within a late-checked block, users should prefer to avoid using operations which, if written outside of the late-checked block and in the nearest enclosing unconstrained context, would bind to a member of a concept map archetype. If said avoidance is not practical, those operations should be clearly documented.

For example, the author of `add(T,T)` should warn other users against satisfying `Semigroup<T>::operator+` with anything other than the `operator+` that would normally be selected for a given type `T`. — *end note*]

4 [*Note:* a late-checked block should be used only when certain suitably constrained versions of templates are not yet available for use from within the body of a constrained template definition and the only viable alternative is the use of an unconstrained template. `late_check` is regarded as an evolutionary tool, to mitigate the cost of migrating a template library to use concepts when the library is based on unconstrained templates that cannot be constrained at that time. The use of `late_check` involves a loss of type-checking and can circumvent the syntax adaptation capabilities provided by concept maps, leading to instantiations that will silently invoke different operations than expected, as in the use of `+` in the `add` example above. For these reasons, the use of `late_check` should be avoided whenever possible. — *end note*]

7 Declarations [dcl.dcl]

1 Declarations specify how names are to be interpreted. Declarations have the form

declaration-seq:
declaration
declaration-seq declaration

declaration:
block-declaration
function-definition
template-declaration
explicit-instantiation
explicit-specialization
linkage-specification
namespace-definition
concept-definition
concept-map-definition
attribute-declaration

block-declaration:
simple-declaration
asm-definition
namespace-alias-definition
using-declaration
using-directive
static_assert-declaration
alias-declaration
opaque-enum-declaration

alias-declaration:
`using` *identifier* = *type-id* `;`

simple-declaration:
attribute-specifier$_{opt}$ *decl-specifier-seq*$_{opt}$ *attribute-specifier*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

static_assert-declaration:
`static_assert (` *constant-expression* `,` *string-literal* `) ;`

attribute-declaration:
attribute-specifier `;`

[*Note:* *asm-definition*s are described in 7.4, and *linkage-specification*s are described in 7.5. *Function-definition*s are described in 8.4 and *template-declaration*s are described in Clause 14. *Namespace-definition*s are described in 7.3.1, *concept-definition*s are described in 14.9.1, *concept-map-definition*s are described in 14.9.2, *using-declaration*s are described in 7.3.3 and *using-directive*s are described in 7.3.4. — *end note*]

The *simple-declaration*

attribute-specifier$_{opt}$ *decl-specifier-seq*$_{opt}$ *attribute-specifier*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

is divided into four parts. *decl-specifier*s, the components of a *decl-specifier-seq*, are described in 7.1. The two optional *attribute-specifier*s and *declarator*s, the components of an *init-declarator-list*, are described in Clause 8.

Except where otherwise specified, the meaning of an *attribute-declaration* is implementation-defined.

2 A declaration occurs in a scope (3.3); the scope rules are summarized in 3.4. A declaration that declares a function or defines a class, concept, concept map, namespace, template, or function also has one or more scopes nested within it. These nested scopes, in turn, can have declarations nested within them. Unless otherwise stated, utterances in Clause 7 about components in, of, or contained by a declaration or subcomponent thereof refer only to those components of the declaration that are *not* nested within scopes nested within the declaration.

3 In a *simple-declaration*, the optional *init-declarator-list* can be omitted only when declaring a class (Clause 9) or enumeration (7.2), that is, when the *decl-specifier-seq* contains either a *class-specifier*, an *elaborated-type-specifier* with a *class-key* (9.1), or an *enum-specifier*. In these cases and whenever a *class-specifier* or *enum-specifier* is present in the *decl-specifier-seq*, the identifiers in these specifiers are among the names being declared by the declaration (as *class-names*, *enum-names*, or *enumerators*, depending on the syntax). In such cases, and except for the declaration of an unnamed bit-field (9.6), the *decl-specifier-seq* shall introduce one or more names into the program, or shall redeclare a name introduced by a previous declaration. [*Example:*

```
enum { };           // ill-formed
typedef class { };  // ill-formed
```

— *end example*]

4 In a *static_assert-declaration* the *constant-expression* shall be a constant expression (5.19) that can be contextually converted to `bool` (Clause 4). If the value of the expression when so converted is `true`, the declaration has no effect. Otherwise, the program is ill-formed, and the resulting diagnostic message (1.4) shall include the text of the *string-literal*, except that characters not in the basic source character set (2.2) are not required to appear in the diagnostic message. [*Example:*

```
static_assert(sizeof(long) >= 8, "64-bit code generation required for this library.");
```

— *end example*]

5 Each *init-declarator* in the *init-declarator-list* contains exactly one *declarator-id*, which is the name declared by that *init-declarator* and hence one of the names declared by the declaration. The *type-specifiers* (7.1.6) in the *decl-specifier-seq* and the recursive *declarator* structure of the *init-declarator* describe a type (8.3), which is then associated with the name being declared by the *init-declarator*.

6 If the *decl-specifier-seq* contains the `typedef` specifier, the declaration is called a *typedef declaration* and the name of each *init-declarator* is declared to be a *typedef-name*, synonymous with its associated type (7.1.3). If the *decl-specifier-seq* contains no `typedef` specifier, the declaration is called a *function declaration* if the type associated with the name is a function type (8.3.5) and an *object declaration* otherwise.

7 Syntactic components beyond those found in the general form of declaration are added to a function declaration to make a *function-definition*. An object declaration, however, is also a definition unless it contains the `extern` specifier and has no initializer (3.1). A definition causes the appropriate amount of storage to be reserved and any appropriate initialization (8.5) to be done.

8 Only in function declarations for constructors, destructors, and type conversions can the *decl-specifier-seq* be omitted.[81] If it is omitted, an *attribute-specifier* shall not appear.

7.1 Specifiers [dcl.spec]

1 The specifiers that can be used in a declaration are

81) The "implicit int" rule of C is no longer supported.

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`friend`
`typedef`
`constexpr`
alignment-specifier

decl-specifier-seq:
decl-specifier-seq$_{opt}$ *decl-specifier*

2 The longest sequence of *decl-specifier*s that could possibly be a type name is taken as the *decl-specifier-seq* of a *declaration*. The sequence shall be self-consistent as described below. [*Example:*

```
typedef char* Pc;
static Pc;                      // error: name missing
```

Here, the declaration `static Pc` is ill-formed because no name was specified for the static variable of type `Pc`. To get a variable called `Pc`, a *type-specifier* (other than `const` or `volatile`) has to be present to indicate that the *typedef-name* `Pc` is the name being (re)declared, rather than being part of the *decl-specifier* sequence. For another example,

```
void f(const Pc);               // void f(char* const) (not const char*)
void g(const int Pc);           // void g(const int)
```

— *end example*]

3 [*Note:* since `signed`, `unsigned`, `long`, and `short` by default imply `int`, a *type-name* appearing after one of those specifiers is treated as the name being (re)declared. [*Example:*

```
void h(unsigned Pc);            // void h(unsigned int)
void k(unsigned int Pc);        // void k(unsigned int)
```

— *end example*] — *end note*]

7.1.1 Storage class specifiers [dcl.stc]

1 The storage class specifiers are

storage-class-specifier:
`register`
`static`
`thread_local`
`extern`
`mutable`

At most one *storage-class-specifier* shall appear in a given *decl-specifier-seq*, except that `thread_local` may appear with `static` or `extern`. If `thread_local` appears in any declaration of an object or reference it shall be present in all declarations of that object or reference. If a *storage-class-specifier* appears in a *decl-specifier-seq*, there can be no `typedef` specifier in the same *decl-specifier-seq* and the *init-declarator-list* of the declaration shall not be empty (except for global anonymous unions, which shall be declared `static` (9.5)). The *storage-class-specifier* applies to the name declared by each *init-declarator* in the list and not to any names declared by other specifiers. A *storage-class-specifier* shall not be specified in an explicit specialization (14.7.3) or an explicit instantiation (14.7.2) directive.

2 The `register` specifier shall be applied only to names of objects declared in a block (6.3) or to function parameters (8.4). It specifies that the named object has automatic storage duration (3.7.3). An object

declared without a *storage-class-specifier* at block scope or declared as a function parameter has automatic storage duration by default.

3 A `register` specifier is a hint to the implementation that the object so declared will be heavily used. [*Note:* the hint can be ignored and in most implementations it will be ignored if the address of the object is taken. — *end note*]

4 The `thread_local` specifier shall be applied only to the names of objects or references of namespace scope and to the names of objects or references of block scope that also specify `static`. It specifies that the named object or reference has thread storage duration (3.7.2).

5 The `static` specifier can be applied only to names of objects and functions and to anonymous unions (9.5). There can be no `static` function declarations within a block, nor any `static` function parameters. A `static` specifier used in the declaration of an object declares the object to have static storage duration (3.7.1), unless accompanied by the `thread_local` specifier, which declares the object to have thread storage duration (3.7.2). A `static` specifier can be used in declarations of class members; 9.4 describes its effect. For the linkage of a name declared with a `static` specifier, see 3.5.

6 The `extern` specifier can be applied only to the names of objects and functions. The `extern` specifier cannot be used in the declaration of class members or function parameters. For the linkage of a name declared with an `extern` specifier, see 3.5. [*Note:* The `extern` keyword can also be used in *explicit-instantiation*s and *linkage-specification*s, but it is not a *storage-class-specifier* in such contexts. — *end note*]

7 A name declared in a namespace scope without a *storage-class-specifier* has external linkage unless it has internal linkage because of a previous declaration and provided it is not declared `const`. Objects declared `const` and not explicitly declared `extern` have internal linkage.

8 The linkages implied by successive declarations for a given entity shall agree. That is, within a given scope, each declaration declaring the same object name or the same overloading of a function name shall imply the same linkage. Each function in a given set of overloaded functions can have a different linkage, however. [*Example:*

```
static char* f();               // f() has internal linkage
char* f()                       // f() still has internal linkage
  { /* ... */ }

char* g();                      // g() has external linkage
static char* g()                // error: inconsistent linkage
  { /* ... */ }

void h();
inline void h();                // external linkage

inline void l();
void l();                       // external linkage

inline void m();
extern void m();                // external linkage

static void n();
inline void n();                // internal linkage

static int a;                   // a has internal linkage
int a;                          // error: two definitions

static int b;                   // b has internal linkage
```

```
extern int b;                   // b still has internal linkage

int c;                          // c has external linkage
static int c;                   // error: inconsistent linkage

extern int d;                   // d has external linkage
static int d;                   // error: inconsistent linkage
```

— *end example*]

9 The name of a declared but undefined class can be used in an `extern` declaration. Such a declaration can only be used in ways that do not require a complete class type. [*Example:*

```
struct S;
extern S a;
extern S f();
extern void g(S);

void h() {
  g(a);                         // error: S is incomplete
  f();                          // error: S is incomplete
}
```

— *end example*]

10 The `mutable` specifier can be applied only to names of class data members (9.2) and cannot be applied to names declared `const` or `static`, and cannot be applied to reference members. [*Example:*

```
class X {
  mutable const int* p;         // OK
  mutable int* const q;         // ill-formed
};
```

— *end example*]

11 The `mutable` specifier on a class data member nullifies a `const` specifier applied to the containing class object and permits modification of the mutable class member even though the rest of the object is `const` (7.1.6.1).

7.1.2 Function specifiers [dcl.fct.spec]

1 *Function-specifiers* can be used only in function declarations.

function-specifier:
`inline`
`virtual`
`explicit`

2 A function declaration (8.3.5, 9.3, 11.4) with an `inline` specifier declares an *inline function*. The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is to be preferred to the usual function call mechanism. An implementation is not required to perform this inline substitution at the point of call; however, even if this inline substitution is omitted, the other rules for inline functions defined by 7.1.2 shall still be respected.

3 A function defined within a class definition is an inline function. The `inline` specifier shall not appear on a block scope function declaration.[82] If the `inline` specifier is used in a friend declaration, that declaration shall be a definition or the function shall have previously been declared inline.

82) The inline keyword has no effect on the linkage of a function.

4 An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case (3.2). [*Note:* a call to the inline function may be encountered before its definition appears in the translation unit. — *end note*] If the definition of a function appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required. An `inline` function with external linkage shall have the same address in all translation units. A `static` local variable in an `extern inline` function always refers to the same object. A string literal in the body of an `extern inline` function is the same object in different translation units. [*Note:* A string literal appearing in a default argument expression is not in the body of an inline function merely because the expression is used in a function call from that inline function. — *end note*]

5 The `virtual` specifier shall be used only in the initial declaration of a non-static class member function; see 10.3.

6 The `explicit` specifier shall be used only in the declaration of a constructor or conversion function within its class definition; see 12.3.1 and 12.3.2.

7.1.3 The `typedef` specifier [dcl.typedef]

1 Declarations containing the *decl-specifier* `typedef` declare identifiers that can be used later for naming fundamental (3.9.1) or compound (3.9.2) types. The `typedef` specifier shall not be used in a *function-definition* (8.4), and it shall not be combined in a *decl-specifier-seq* with any other kind of specifier except a *type-specifier*.

typedef-name:
identifier

A name declared with the `typedef` specifier becomes a *typedef-name*. Within the scope of its declaration, a *typedef-name* is syntactically equivalent to a keyword and names the type associated with the identifier in the way described in Clause 8. A *typedef-name* is thus a synonym for another type. A *typedef-name* does not introduce a new type the way a class declaration (9.1) or enum declaration does. [*Example:* after

```
typedef int MILES, *KLICKSP;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
```

are all correct declarations; the type of `distance` is `int` and that of `metricp` is "pointer to `int`." — *end example*]

2 A *typedef-name* can also be introduced by an *alias-declaration*. The *identifier* following the `using` keyword becomes a *typedef-name*. It has the same semantics as if it were introduced by the `typedef` specifier. In particular, it does not define a new type and it shall not appear in the *type-id*. [*Example:*

```
using handler_t = void (*)(int);
extern handler_t ignore;
extern void (*ignore)(int);          // redeclare ignore
using cell = pair<void*, cell*>;     // ill-formed
```

— *end example*]

3 In a given non-class scope, a `typedef` specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers. [*Example:*

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

— *end example*]

4 In a given class scope, a `typedef` specifier can be used to redefine any *class-name* declared in that scope that is not also a *typedef-name* to refer to the type to which it already refers. [*Example:*

```
struct S {
  typedef struct A { } A;       // OK
  typedef struct B B;           // OK
  typedef A A;                  // error
```

— *end example*]

5 In a given scope, a `typedef` specifier shall not be used to redefine the name of any type declared in that scope to refer to a different type. [*Example:*

```
class complex { /* ... */ };
typedef int complex;            // error: redefinition
```

— *end example*]

6 Similarly, in a given scope, a class or enumeration shall not be declared with the same name as a *typedef-name* that is declared in that scope and refers to a type other than the class or enumeration itself. [*Example:*

```
typedef int complex;
class complex { /* ...  */ };   // error: redefinition
```

— *end example*]

7 [*Note:* A *typedef-name* that names a class type, or a cv-qualified version thereof, is also a *class-name* (9.1). If a *typedef-name* is used to identify the subject of an *elaborated-type-specifier* (7.1.6.3), a class definition (Clause 9), a constructor declaration (12.1), or a destructor declaration (12.4), the program is ill-formed. — *end note*] [*Example:*

```
struct S {
  S();
  ~S();
};

typedef struct S T;

S a = T();                      // OK
struct T * p;                   // error
```

— *end example*]

8 If the typedef declaration defines an unnamed class (or enum), the first *typedef-name* declared by the declaration to be that class type (or enum type) is used to denote the class type (or enum type) for linkage purposes only (3.5). [*Example:*

```
typedef struct { } *ps, S;      // S is the class name for linkage purposes
```

— *end example*]

9 If a typedef `TD` names a type that is a reference to a type `T`, an attempt to create the type "lvalue reference to *cv* `TD`" creates the type "lvalue reference to `T`," while an attempt to create the type "rvalue reference to *cv* `TD`" creates the type `TD`. [*Example:*

```
int i;
typedef int& LRI;
typedef int&& RRI;

LRI& r1 = i;                    // r1 has the type int&
const LRI& r2 = i;              // r2 has the type int&
const LRI&& r3 = i;             // r3 has the type int&

RRI& r4 = i;                    // r4 has the type int&
RRI&& r5 = i;                   // r5 has the type int&&
```

— *end example*]

7.1.4 The `friend` specifier [dcl.friend]

The `friend` specifier is used to specify access to class members; see 11.4.

7.1.5 The `constexpr` specifier [dcl.constexpr]

1 The `constexpr` specifier shall be applied only to the definition of an object, function, or function template, or to the declaration of a static data member of an effective literal type (3.9). [*Note:* function parameters cannot be declared `constexpr`. — *end note*] [*Example:*

```
constexpr int square(int x) {   // OK
  return x * x;
}
constexpr int bufsz = 1024;     // OK
constexpr struct pixel {        // error: pixel is a type
  int x;
  int y;
};
int next(constexpr int x) {     // error
     return x + 1;
}
extern constexpr int memsz;     // error: not a definition
```

— *end example*]

2 A `constexpr` specifier used in the declaration of a function that is not a constructor declares that function to be a *constexpr function*. Similarly, a `constexpr` specifier used in a constructor declaration declares that constructor to be a *constexpr constructor*. Constexpr functions and constexpr constructors are implicitly `inline` (7.1.2).

3 The definition of a constexpr function shall satisfy the following constraints:

— it shall not be virtual (10.3)

— its return type shall be an effective literal type

— each of its parameter types shall be an effective literal type

— its *function-body* shall be a *compound-statement* of the form

```
{ return expression; }
```

where *expression* is a potential constant expression (5.19)

— every implicit conversion used in converting *expression* to the function return type (8.5) shall be one of those allowed in a constant expression (5.19).

[*Example:*

```
constexpr int square(int x)
  { return x * x; }             // OK
constexpr long long_max()
  { return 2147483647; }        // OK
constexpr int abs(int x)
  { return x < 0 ? -x : x; }    // OK
constexpr void f(int x)         // error: return type is void
  { /* ... */ }
constexpr int prev(int x)
  { return --x; }               // error: use of decrement
constexpr int g(int x, int n) { // error: body not just "return expr"
  int r = 1;
  while (--n > 0) r *= x;
  return r;
}
```

— *end example*]

4 The definition of a constexpr constructor shall satisfy the following constraints:

— each of its parameter types shall be an effective literal type

— its *function-body* shall not be a *function-try-block*

— the *compound-statement* of its *function-body* shall be empty

— every non-static data member and base class sub-object shall be initialized (12.6.2)

— every constructor involved in initializing non-static data members and base class sub-objects invoked by a *mem-initializer* shall be a constexpr constructor.

— every constructor argument and full-expression in a *mem-initializer* shall be a potential constant expression

— every implicit conversion used in converting a constructor argument to the corresponding parameter type and converting a full-expression to the corresponding member type shall be one of those allowed in a constant expression.

A trivial copy constructor is also a constexpr constructor.

[*Example:*

```
struct Length {
  explicit constexpr Length(int i = 0) : val(i) { }
private:
    int val;
};
```

— *end example*]

5 If the instantiated template specialization of a constexpr function template would fail to satisfy the requirements for a constexpr function or constexpr constructor, the `constexpr` specifier is ignored..

6 A `constexpr` specifier for a non-static member function that is not a constructor declares that member function to be `const` (9.3.1). [*Note:* the `constexpr` specifier has no other effect on the function type. — *end note*] The class of which that function is a member shall be a literal type (3.9). [*Example:*

```
class debug_flag {
public:
  explicit debug_flag(bool);
  constexpr bool is_on();       // error: debug_flag not
                                // literal type
private:
  bool flag;
};
constexpr int bar(int x, int y) // OK
    { return x + y + x*y; }
// ...
int bar(int x, int y)           // error: redefinition of bar
    { return x * 2 + 3 * y; }
```

— *end example*]

7 A `constexpr` specifier used in an object declaration declares the object as `const`. Such an object shall be initialized. If it is initialized by a constructor call, the constructor shall be a constexpr constructor and every argument to the constructor shall be a constant expression. Otherwise, every full-expression that appears in its initializer shall be a constant expression. Each implicit conversion used in converting the initializer expressions and each constructor call used for the initialization shall be one of those allowed in a constant expression (5.19). [*Example:*

```
struct pixel {
  int x, y;
};
constexpr pixel ur = { 1294, 1024 };// OK
constexpr pixel origin;             // error: initializer missing
```

— *end example*]

7.1.6 Type specifiers [dcl.type]

1 The type-specifiers are

type-specifier:
simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
typename-specifier
cv-qualifier

type-specifier-seq:
type-specifier type-specifier-seq$_{opt}$

2 As a general rule, at most one *type-specifier* is allowed in the complete *decl-specifier-seq* of a *declaration* or in a *type-specifier-seq*. The only exceptions to this rule are the following:

— `const` can be combined with any type specifier except itself.

— `volatile` can be combined with any type specifier except itself.

— `signed` or `unsigned` can be combined with `char`, `long`, `short`, or `int`.

— `short` or `long` can be combined with `int`.

— `long` can be combined with `double`.

— `long` can be combined with `long`.

3 At least one *type-specifier* that is not a *cv-qualifier* is required in a declaration unless it declares a constructor, destructor or conversion function.[83] A *type-specifier-seq* shall not define a class or enumeration unless it appears in the *type-id* of an *alias-declaration* (7.1.3).

4 [*Note: class-specifier*s and *enum-specifier*s are discussed in Clause 9 and 7.2, respectively. The remaining *type-specifier*s are discussed in the rest of this section. — *end note*]

7.1.6.1 The *cv-qualifiers* [dcl.type.cv]

1 There are two *cv-qualifiers*, `const` and `volatile`. If a *cv-qualifier* appears in a *decl-specifier-seq*, the *init-declarator-list* of the declaration shall not be empty. [*Note:* 3.9.3 describes how cv-qualifiers affect object and function types. — *end note*] Redundant cv-qualifications are ignored. [*Note:* for example, these could be introduced by typedefs. — *end note*]

2 An object declared in namespace scope with a const-qualified type has internal linkage unless it is explicitly declared `extern` or unless it was previously declared to have external linkage. A variable of non-volatile const-qualified integral or enumeration type initialized by an integral constant expression can be used in integral constant expressions (5.19). [*Note:* as described in 8.5, the definition of an object or subobject of const-qualified type must specify an initializer or be subject to default-initialization. — *end note*]

3 A pointer or reference to a cv-qualified type need not actually point or refer to a cv-qualified object, but it is treated as if it does; a const-qualified access path cannot be used to modify an object even if the object referenced is a non-const object and can be modified through some other access path. [*Note:* cv-qualifiers are supported by the type system so that they cannot be subverted without casting (5.2.11). — *end note*]

4 Except that any class member declared `mutable` (7.1.1) can be modified, any attempt to modify a `const` object during its lifetime (3.8) results in undefined behavior. [*Example:*

```
const int ci = 3;               // cv-qualified (initialized as required)
ci = 4;                         // ill-formed: attempt to modify const

int i = 2;                      // not cv-qualified
const int* cip;                 // pointer to const int
cip = &i;                       // OK: cv-qualified access path to unqualified
*cip = 4;                       // ill-formed: attempt to modify through ptr to const

int* ip;
ip = const_cast<int*>(cip);     // cast needed to convert const int* to int*
*ip = 4;                        // defined: *ip points to i, a non-const object

const int* ciq = new const int (3);     // initialized as required
int* iq = const_cast<int*>(ciq);        // cast required
*iq = 4;                                // undefined: modifies a const object
```

5 For another example

```
struct X {
  mutable int i;
```

83) There is no special provision for a *decl-specifier-seq* that lacks a *type-specifier* or that has a *type-specifier* that only specifies *cv-qualifier*s. The "implicit int" rule of C is no longer supported.

```
  int j;
};
struct Y {
  X x;
  Y();
};

const Y y;
y.x.i++;                        // well-formed: mutable member can be modified
y.x.j++;                        // ill-formed: const-qualified member modified
Y* p = const_cast<Y*>(&y);      // cast away const-ness of y
p->x.i = 99;                    // well-formed: mutable member can be modified
p->x.j = 99;                    // undefined: modifies a const member
```

— *end example*]

6 If an attempt is made to refer to an object defined with a volatile-qualified type through the use of an lvalue with a non-volatile-qualified type, the program behavior is undefined.

7 [*Note:* `volatile` is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of `volatile` are intended to be the same in C++ as they are in C. — *end note*]

7.1.6.2 Simple type specifiers [dcl.type.simple]

1 The simple type specifiers are

simple-type-specifier:
- ::$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
- ::$_{opt}$ *nested-name-specifier* `template` *simple-template-id*
- `char`
- `char16_t`
- `char32_t`
- `wchar_t`
- `bool`
- `short`
- `int`
- `long`
- `signed`
- `unsigned`
- `float`
- `double`
- `void`
- `auto`
- `decltype (` *expression* `)`

type-name:
- *class-name*
- *enum-name*
- *typedef-name*

2 The `auto` specifier is a placeholder for a type to be deduced (7.1.6.4). The other *simple-type-specifier*s specify either a previously-declared user-defined type or one of the fundamental types (3.9.1). Table 9 summarizes the valid combinations of *simple-type-specifier*s and the types they specify.

Table 9 — *simple-type-specifier*s and the types they specify

Specifier(s)	Type
type-name	the type named
char	"char"
unsigned char	"unsigned char"
signed char	"signed char"
char16_t	"char16_t"
char32_t	"char32_t"
bool	"bool"
unsigned	"unsigned int"
unsigned int	"unsigned int"
signed	"int"
signed int	"int"
int	"int"
unsigned short int	"unsigned short int"
unsigned short	"unsigned short int"
unsigned long int	"unsigned long int"
unsigned long	"unsigned long int"
unsigned long long int	"unsigned long long int"
unsigned long long	"unsigned long long int"
signed long int	"long int"
signed long	"long int"
signed long long int	"long long int"
signed long long	"long long int"
long long int	"long long int"
long long	"long long int"
long int	"long int"
long	"long int"
signed short int	"short int"
signed short	"short int"
short int	"short int"
short	"short int"
wchar_t	"wchar_t"
float	"float"
double	"double"
long double	"long double"
void	"void"
auto	placeholder for a type to be deduced
decltype(*expression*)	the type as defined below

3 When multiple *simple-type-specifiers* are allowed, they can be freely intermixed with other *decl-specifiers* in any order. [*Note:* It is implementation-defined whether objects of `char` type and certain bit-fields (9.6) are represented as signed or unsigned quantities. The `signed` specifier forces `char` objects and bit-fields to be signed; it is redundant in other contexts. — *end note*]

4 The type denoted by `decltype(e)` is defined as follows:

— if `e` is an *id-expression* or a class member access (5.2.5), `decltype(e)` is the type of the entity named by `e`. If there is no such entity, or if `e` names a set of overloaded functions, the program is ill-formed;

— otherwise, if `e` is a function call (5.2.2) or an invocation of an overloaded operator (parentheses around `e` are ignored), `decltype(e)` is the return type of the statically chosen function;

— otherwise, if `e` is an lvalue, `decltype(e)` is `T&`, where `T` is the type of `e`;

— otherwise, `decltype(e)` is the type of `e`.

The operand of the `decltype` specifier is an unevaluated operand (Clause 5).

[*Example:*

```
const int&& foo();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1;             // type is const int&&
decltype(i) x2;                 // type is int
decltype(a->x) x3;              // type is double
decltype((a->x)) x4;            // type is const double&
```

— *end example*]

7.1.6.3 Elaborated type specifiers [dcl.type.elab]

elaborated-type-specifier:
class-key ::$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
class-key ::$_{opt}$ *nested-name-specifier*$_{opt}$ `template`$_{opt}$ *simple-template-id*
enum ::$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

1 If an *elaborated-type-specifier* is the sole constituent of a declaration, the declaration is ill-formed unless it is an explicit specialization (14.7.3), an explicit instantiation (14.7.2) or it has one of the following forms:

class-key identifier attribute-specifier$_{opt}$;
`friend` *class-key* ::$_{opt}$ *identifier* ;
`friend` *class-key* ::$_{opt}$ *simple-template-id* ;
`friend` *class-key* ::$_{opt}$ *nested-name-specifier identifier* ;
`friend` *class-key* ::$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *simple-template-id* ;

In the first case, the *attribute-specifier*, if any, appertains to the class being declared; the attributes in the *attribute-specifier* are thereafter considered attributes of the class whenever it is named.

2 3.4.4 describes how name lookup proceeds for the *identifier* in an *elaborated-type-specifier*. If the *identifier* resolves to a *class-name* or *enum-name*, the *elaborated-type-specifier* introduces it into the declaration the same way a *simple-type-specifier* introduces its *type-name*. If the *identifier* resolves to a *typedef-name*, the *elaborated-type-specifier* is ill-formed. [*Note:* this implies that, within a class template with a template *type-parameter* `T`, the declaration

```
friend class T;
```

is ill-formed. — *end note*]

3 The *class-key* or `enum` keyword present in the *elaborated-type-specifier* shall agree in kind with the declaration to which the name in the *elaborated-type-specifier* refers. This rule also applies to the form of *elaborated-type-specifier* that declares a *class-name* or `friend` class since it can be construed as referring to the definition of the class. Thus, in any *elaborated-type-specifier*, the `enum` keyword shall be used to refer to an enumeration (7.2), the `union` *class-key* shall be used to refer to a union (Clause 9), and either the `class` or `struct` *class-key* shall be used to refer to a class (Clause 9) declared using the `class` or `struct` *class-key*. [*Example:*

```
enum class E { a, b };
enum E x = E::a;                // OK
```

— *end example*]

7.1.6.4 `auto` specifier [dcl.spec.auto]

1 The `auto` *type-specifier* signifies that the type of an object being declared shall be deduced from its initializer or specified explicitly at the end of a function declarator.

2 The `auto` *type-specifier* may appear with a function declarator with a late-specified return type (8.3.5) in any context where such a declarator is valid, and the use of `auto` is replaced by the type specified at the end of the declarator.

3 Otherwise, the type of the object is deduced from its initializer. The name of the object being declared shall not appear in the initializer expression. This use of `auto` is allowed when declaring objects in a block (6.3), in namespace scope (3.3.5), and in a *for-init-statement* (6.5.3). The *decl-specifier-seq* shall be followed by one or more *init-declarator*s, each of which shall have a non-empty *initializer* of either of the following forms:

`=` *assignment-expression*
`(` *assignment-expression* `)`

[*Example:*

```
auto x = 5;                 // OK: x has type int
const auto *v = &x, u = 6;  // OK: v has type const int*, u has type const int
static auto y = 0.0;        // OK: y has type double
auto int r;                 // error: auto is not a storage-class-specifier
```

— *end example*]

4 The `auto` *type-specifier* can also be used in declaring an object in the *condition* of a selection statement (6.4) or an iteration statement (6.5), in the *type-specifier-seq* in a *new-type-id* (5.3.4), in a *for-range-declaration*, and in declaring a static data member with a *brace-or-equal-initializer* that appears within the *member-specification* of a class definition (9.4.2).

5 A program that uses `auto` in a context not explicitly allowed in this section is ill-formed.

6 Once the type of a *declarator-id* has been determined according to 8.3, the type of the declared variable using the *declarator-id* is determined from the type of its initializer using the rules for template argument deduction. Let `T` be the type that has been determined for a variable identifier `d`. Obtain `P` from `T` by replacing the occurrences of `auto` with either a new invented type template parameter `U` or, if the initializer is a *braced-init-list* (8.5.4), with `std::initializer_list<U>`. The type deduced for the variable `d` is then the deduced type determined using the rules of template argument deduction from a function call (14.8.2.1), where `P` is a function template parameter type and the initializer for `d` is the corresponding argument. If the deduction fails, the declaration is ill-formed. [*Example:*

```
auto x1 = { 1, 2 };         // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 };       // error: cannot deduce element type
```

— *end example*]

7 If the list of declarators contains more than one declarator, the type of each declared variable is determined as described above. If the type deduced for the template parameter `U` is not the same in each deduction, the program is ill-formed.

[*Example:*

```
const auto &i = expr;
```

The type of `i` is the deduced type of the parameter `u` in the call `f(expr)` of the following invented function template:

```
template <class U> void f(const U& u);
```

— *end example*]

7.2 Enumeration declarations [dcl.enum]

1 An enumeration is a distinct type (3.9.1) with named constants. Its name becomes an *enum-name*, within its scope.

enum-name:
identifier

enum-specifier:
enum-head `{` *enumerator-list*$_{opt}$ `}`
enum-head `{` *enumerator-list* `,` `}`

enum-head:
enum-key identifier$_{opt}$ *attribute-specifier*$_{opt}$ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$
enum-key nested-name-specifier identifier
attribute-specifier$_{opt}$ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$

opaque-enum-declaration:
enum-key identifier attribute-specifier$_{opt}$ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$ `;`

enum-key:
`enum`
`enum class`
`enum struct`

enum-base:
`:` *type-specifier-seq*

enumerator-list:
enumerator-definition
enumerator-list `,` *enumerator-definition*

enumerator-definition:
enumerator
enumerator `=` *constant-expression*

enumerator:
identifier

The first optional *attribute-specifier* in the *enum-head* and the *opaque-enum-declaration* appertains to the enumeration; the attributes in that *attribute-specifier* are thereafter considered attributes of the enumeration whenever it is named. The second optional *attribute-specifier* in the *enum-head* and the *opaque-enum-declaration* shall appear only if the *enum-base* is present; it appertains to the *enum-base*.

2 The enumeration type declared with an *enum-key* of only `enum` is an *unscoped enumeration*, and its *enumerator*s are *unscoped enumerators*. The *enum-key*s `enum class` and `enum struct` are semantically equivalent; an enumeration type declared with one of these is a *scoped enumeration*, and its *enumerator*s are *scoped enumerators*. The optional *identifier* shall not be omitted in the declaration of a scoped enumeration. The *type-specifier-seq* of an *enum-base* shall name an integral type; any cv-qualification is ignored. An *opaque-enum-declaration* declaring an unscoped enumeration shall not omit the *enum-base*. The identifiers in an *enumerator-list* are declared as constants, and can appear wherever constants are required. An *enumerator-definition* with `=` gives the associated *enumerator* the value indicated by the *constant-expression*. The *constant-expression* shall be an integral constant expression (5.19). If the first *enumerator* has no *initializer*, the value of the corresponding constant is zero. An *enumerator-definition* without an *initializer* gives the *enumerator* the value obtained by increasing the value of the previous *enumerator* by one.

[*Example:*

```
enum { a, b, c=0 };
enum { d, e, f=e+2 };
```

defines `a`, `c`, and `d` to be zero, `b` and `e` to be `1`, and `f` to be `3`. — *end example*]

3 An *opaque-enum-declaration* is either a redeclaration of an enumeration in the current scope or a declaration of a new enumeration. [*Note:* an enumeration declared by an *opaque-enum-declaration* has fixed underlying type and is a complete type. The list of enumerators can be provided in a later redeclaration with an *enum-specifier*. — *end note*] A scoped enumeration shall not be later redeclared as unscoped or with a different underlying type. An unscoped enumeration shall not be later redeclared as scoped and each redeclaration shall include an *enum-base* specifying the same underlying type.

4 If the *enum-key* is followed by a *nested-name-specifier*, the *enum-specifier* shall refer to an enumeration that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers (i.e., neither inherited nor introduced by a *using-declaration*), and the *enum-specifier* shall appear in a namespace enclosing the previous declaration.

5 Each enumeration defines a type that is different from all other types. Each enumeration also has an *underlying type*. The underlying type can be explicitly specified using *enum-base*; if not explicitly specified, the underlying type of a scoped enumeration type is `int`. In these cases, the underlying type is said to be *fixed*. Following the closing brace of an *enum-specifier*, each enumerator has the type of its enumeration. If the underlying type is fixed, the type of each *enumerator* prior to the closing brace is the underlying type; if the initializing value of an *enumerator* cannot be represented by the underlying type, the program is ill-formed. If the underlying type is not fixed, the type of each enumerator is the type of its initializing value:

— If an initializer is specified for an enumerator, the initializing value has the same type as the expression.

— If no initializer is specified for the first enumerator, the initializing value has an unspecified integral type.

— Otherwise the type of the initializing value is the same as the type of the initializing value of the preceding enumerator unless the incremented value is not representable in that type, in which case the type is an unspecified integral type sufficient to contain the incremented value.

6 For an enumeration whose underlying type is not fixed, the underlying type is an integral type that can represent all the enumerator values defined in the enumeration. If no integral type can represent all the enumerator values, the enumeration is ill-formed. It is implementation-defined which integral type is used as the underlying type except that the underlying type shall not be larger than `int` unless the value of an enumerator cannot fit in an `int` or `unsigned int`. If the *enumerator-list* is empty, the underlying type is as if the enumeration had a single enumerator with value 0.

7 For an enumeration whose underlying type is fixed, the values of the enumeration are the values of the underlying type. Otherwise, for an enumeration where e_{min} is the smallest enumerator and e_{max} is the largest, the values of the enumeration are the values in the range b_{min} to b_{max}, defined as follows: Let K be 1 for a two's complement representation and 0 for a one's complement or sign-magnitude representation. b_{max} is the smallest value greater than or equal to $max(|e_{min}| - K, |e_{max}|)$ and equal to $2^M - 1$, where M is a non-negative integer. b_{min} is zero if e_{min} is non-negative and $-(b_{max} + K)$ otherwise. The size of the smallest bit-field large enough to hold all the values of the enumeration type is $max(M, 1)$ if b_{min} is zero and $M + 1$ otherwise. It is possible to define an enumeration that has values not defined by any of its enumerators.

8 Two enumeration types are layout-compatible if they have the same *underlying type*.

9 The value of an enumerator or an object of an unscoped enumeration type is converted to an integer by integral promotion (4.5). [*Example:*

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue)              // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue` these values can be converted to the integral values `0`, `1`, `20`, and `21`. Since enumerations are distinct types, objects of type `color` can be assigned only values of type `color`.

```
color c = 1;                  // error: type mismatch,
                              // no conversion from int to color

int i = yellow;               // OK: yellow converted to integral value 1
                              // integral promotion
```

Note that this implicit `enum` to `int` conversion is not provided for a scoped enumeration:

```
enum class Col { red, yellow, green };
int x = Col::red;             // error: no Col to int conversion
Col y = Col::red;
if (y) { }                    // error: no Col to bool conversion
```

— *end example*]

10 An expression of arithmetic or enumeration type can be converted to an enumeration type explicitly. The value is unchanged if it is in the range of enumeration values of the enumeration type; otherwise the resulting enumeration value is unspecified.

11 Each *enum-name* and each unscoped *enumerator* is declared in the scope that immediately contains the *enum-specifier*. Each scoped *enumerator* is declared in the scope of the enumeration. These names obey the scope rules defined for all names in (3.3) and (3.4).[*Example:*

```
enum direction { left='l', right='r' };

void g()  {
  direction d;                // OK
  d = left;                   // OK
  d = direction::right;       // OK
}

enum class altitude { high='h', low='l' };

void h()  {
  altitude a;                 // OK
  a = high;                   // error: high not in scope
  a = altitude::low;          // OK
}
```

— *end example*] An enumerator declared in class scope can be referred to using the class member access operators (`::`, `.` (dot) and `->` (arrow)), see 5.2.5. [*Example:*

```
struct X {
  enum direction { left='l', right='r' };
```

```
  int f(int i) { return i==left ? 0 : i==right ? 1 : 2; }
};

void g(X* p) {
  direction d;                  // error: direction not in scope
  int i;
  i = p->f(left);               // error: left not in scope
  i = p->f(X::right);           // OK
  i = p->f(p->left);            // OK
  // ...
}
```

— *end example*]

7.3 Namespaces [basic.namespace]

1 A namespace is an optionally-named declarative region. The name of a namespace can be used to access entities declared in that namespace; that is, the members of the namespace. Unlike other declarative regions, the definition of a namespace can be split over several parts of one or more translation units.

2 The outermost declarative region of a translation unit is a namespace; see 3.3.5.

7.3.1 Namespace definition [namespace.def]

1 The grammar for a *namespace-definition* is

namespace-name:
    original-namespace-name
    namespace-alias
original-namespace-name:
    identifier
namespace-definition:
    named-namespace-definition
    unnamed-namespace-definition
named-namespace-definition:
    original-namespace-definition
    extension-namespace-definition
original-namespace-definition:
    inline$_{opt}$ **namespace** *identifier* **{** *namespace-body* **}**
extension-namespace-definition:
    inline$_{opt}$ **namespace** *original-namespace-name* **{** *namespace-body* **}**
unnamed-namespace-definition:
    inline$_{opt}$ **namespace** **{** *namespace-body* **}**
namespace-body:
    declaration-seq$_{opt}$

2 The *identifier* in an *original-namespace-definition* shall not have been previously defined in the declarative region in which the *original-namespace-definition* appears. The *identifier* in an *original-namespace-definition* is the name of the namespace. Subsequently in that declarative region, it is treated as an *original-namespace-name*.

3 The *original-namespace-name* in an *extension-namespace-definition* shall have previously been defined in an *original-namespace-definition* in the same declarative region.

4 Every *namespace-definition* shall appear in the global scope or in a namespace scope (3.3.5).

5 Because a *namespace-definition* contains *declarations* in its *namespace-body* and a *namespace-definition* is itself a *declaration*, it follows that *namespace-definitions* can be nested. [*Example:*

```
namespace Outer {
  int i;
  namespace Inner {
    void f() { i++; }           // Outer::i
    int i;
    void g() { i++; }           // Inner::i
  }
}
```

— *end example*]

6 The *enclosing namespaces* of a declaration are those namespaces in which the declaration lexically appears, except for a redeclaration of a namespace member outside its original namespace (e.g., a definition as specified in 7.3.1.2). Such a redeclaration has the same enclosing namespaces as the original declaration. [*Example:*

```
namespace Q {
  namespace V {
    void f();   // enclosing namespaces are the global namespace, Q, and Q::V
    class C { void m(); };
  }
  void V::f() { // enclosing namespaces are the global namespace, Q, and Q::V
    extern void h();  // ... so this declares Q::V::h
  }
  void V::C::m() { // enclosing namespaces are the global namespace, Q, and Q::V
  }
}
```

— *end example*]

7 If the optional initial `inline` keyword appears in a *namespace-definition* for a particular namespace, that namespace is declared to be an *inline namespace*. The `inline` keyword may be used on an *extension-namespace-definition* only if it was previously used on the *original-namespace-definition* for that namespace.

8 Members of an inline namespace can be used in most respects as though they were members of the enclosing namespace. Specifically, the inline namespace and its enclosing namespace are considered to be associated namespaces (3.4.2) of one another, and a *using-directive* (7.3.4) that names the inline namespace is implicitly inserted into the enclosing namespace. Furthermore, each member of the inline namespace can subsequently be explicitly instantiated (14.7.2) or explicitly specialized (14.7.3) as though it were a member of the enclosing namespace. Finally, looking up a name in the enclosing namespace via explicit qualification (3.4.3.2) will include members of the inline namespace brought in by the *using-directive* even if there are declarations of that name in the enclosing namespace.

9 These properties are transitive: if a namespace `N` contains an inline namespace `M`, which in turn contains an inline namespace `O`, then the members of `O` can be used as though they were members of `M` or `N`. The set of namespaces consisting of the innermost non-inline namespace enclosing an inline namespace `O`, together with any intervening inline namespaces, is the *enclosing namespace set* of `O`.

7.3.1.1 Unnamed namespaces [namespace.unnamed]

1 An *unnamed-namespace-definition* behaves as if it were replaced by

```
namespace unique { /* empty body */ }
using namespace unique ;
namespace unique { namespace-body }
```

where all occurrences of ***unique*** in a translation unit are replaced by the same identifier and this identifier differs from all other identifiers in the entire program.[84] [*Example:*

```
namespace { int i; }            // unique ::i
void f() { i++; }               // unique ::i++

namespace A {
  namespace {
    int i;                      // A:: unique ::i
    int j;                      // A:: unique ::j
  }
  void g() { i++; }             // A:: unique ::i++
}

using namespace A;
void h() {
  i++;                          // error: unique ::i or A:: unique ::i
  A::i++;                       // A:: unique ::i
  j++;                          // A:: unique ::j
}
```

— *end example*]

2 The use of the `static` keyword is deprecated when declaring objects in a namespace scope (see annex D); the *unnamed-namespace* provides a superior alternative.

7.3.1.2 Namespace member definitions [namespace.memdef]

1 Members (including explicit specializations of templates (14.7.3)) of a namespace can be defined within that namespace. [*Example:*

```
namespace X {
  void f() { /* ... */ }
}
```

— *end example*]

2 Members (including explicit specializations of templates (14.7.3)) of a named namespace can also be defined outside that namespace by explicit qualification (3.4.3.2) of the name being defined, provided that the entity being defined was already declared in the namespace and the definition appears after the point of declaration in a namespace that encloses the declaration's namespace. [*Example:*

```
namespace Q {
  namespace V {
    void f();
  }
  void V::f() { /* ... */ }     // OK
  void V::g() { /* ... */ }     // error: g() is not yet a member of V
  namespace V {
    void g();
  }
}

namespace R {
```

84) Although entities in an unnamed namespace might have external linkage, they are effectively qualified by a name unique to their translation unit and therefore can never be seen from any other translation unit.

```
  void Q::V::g() { /* ... */ }  // error: R doesn't enclose Q
}
```

— *end example*]

3 Every name first declared in a namespace is a member of that namespace. If a `friend` declaration in a non-local class first declares a class or function[85] the friend class or function is a member of the innermost enclosing namespace. The name of the friend is not found by unqualified lookup (3.4.1) or by qualified lookup (3.4.3) until a matching declaration is provided in that namespace scope (either before or after the class definition granting friendship). If a friend function is called, its name may be found by the name lookup that considers functions from namespaces and classes associated with the types of the function arguments (3.4.2). If the name in a `friend` declaration is neither qualified nor a *template-id* and the declaration is a function or an *elaborated-type-specifier*, the lookup to determine whether the entity has been previously declared shall not consider any scopes outside the innermost enclosing namespace. [*Note:* the other forms of `friend` declarations cannot declare a new member of the innermost enclosing namespace and thus follow the usual lookup rules. — *end note*] [*Example:*

```
// Assume f and g have not yet been defined.
void h(int);
template <class T> void f2(T);
namespace A {
  class X {
    friend void f(X);           // A::f(X) is a friend
    class Y {
      friend void g();          // A::g is a friend
      friend void h(int);       // A::h is a friend
                                // ::h not considered
      friend void f2<>(int);    // ::f2<>(int) is a friend
    };
  };

  // A::f, A::g and A::h are not visible here
  X x;
  void g() { f(x); }            // definition of A::g
  void f(X) { /* ...  */}       // definition of A::f
  void h(int) { /* ...  */ }    // definition of A::h
  // A::f, A::g and A::h are visible here and known to be friends
}

using A::x;

void h() {
  A::f(x);
  A::X::f(x);                   // error: f is not a member of A::X
  A::X::Y::g();                 // error: g is not a member of A::X::Y
}
```

— *end example*]

7.3.2 Namespace alias [namespace.alias]

1 A *namespace-alias-definition* declares an alternate name for a namespace according to the following grammar:

85) this implies that the name of the class or function is unqualified.

namespace-alias:
identifier
namespace-alias-definition:
`namespace` *identifier* `=` *qualified-namespace-specifier* `;`
qualified-namespace-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name*

2 The *identifier* in a *namespace-alias-definition* is a synonym for the name of the namespace denoted by the *qualified-namespace-specifier* and becomes a *namespace-alias*. [*Note:* when looking up a *namespace-name* in a *namespace-alias-definition*, only namespace names are considered, see 3.4.6. — *end note*]

3 In a declarative region, a *namespace-alias-definition* can be used to redefine a *namespace-alias* declared in that declarative region to refer only to the namespace to which it already refers. [*Example:* the following declarations are well-formed:

```
namespace Company_with_very_long_name { /* ... */ }
namespace CWVLN = Company_with_very_long_name;
namespace CWVLN = Company_with_very_long_name;          // OK: duplicate
namespace CWVLN = CWVLN;
```

— *end example*]

4 A *namespace-name* or *namespace-alias* shall not be declared as the name of any other entity in the same declarative region. A *namespace-name* defined at global scope shall not be declared as the name of any other entity in any global scope of the program. No diagnostic is required for a violation of this rule by declarations in different translation units.

7.3.3 The `using` declaration [namespace.udecl]

1 A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears.

using-declaration:
`using` `typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using` `::` *unqualified-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name*$_{opt}$ `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `;`

The member name specified in a *using-declaration* is declared in the declarative region in which the *using-declaration* appears. [*Note:* only the specified name is so declared; specifying an enumeration name in a *using-declaration* does not declare its enumerators in the *using-declaration*'s declarative region. — *end note*] If a *using-declaration* names a constructor (3.4.3.1), it implicitly declares a set of constructors in the class in which the *using-declaration* appears (12.9); otherwise the name specified in a *using-declaration* is a synonym for the name of some entity declared elsewhere.

2 Every *using-declaration* is a *declaration* and a *member-declaration* and so can be used in a class definition. [*Example:*

```
struct B {
  void f(char);
  void g(char);
  enum E { e };
  union { int x; };
};

struct D : B {
  using B::f;
  void f(int) { f('c'); }       // calls B::f(char)
```

```
  void g(int) { g('c'); }       // recursively calls D::g(int)
};
```

— *end example*]

3 In a *using-declaration* used as a *member-declaration*, the *nested-name-specifier* shall name a base class of the class being defined. If such a *using-declaration* names a constructor, the *nested-name-specifier* shall name a direct base class of the class being defined; otherwise it introduces the set of declarations found by member name lookup (10.2, 3.4.3.1). [*Example:*

```
class C {
  int g();
};

class D2 : public B {
  using B::f;                   // OK: B is a base of D2
  using B::e;                   // OK: e is an enumerator of base B
  using B::x;                   // OK: x is a union member of base B
  using C::g;                   // error: C isn't a base of D2
};
```

— *end example*]

4 [*Note:* Since destructors do not have names, a *using-declaration* cannot refer to a destructor for a base class. Since specializations of member templates for conversion functions are not found by name lookup, they are not considered when a *using-declaration* specifies a conversion function (14.5.2). — *end note*] If an assignment operator brought from a base class into a derived class scope has the signature of a copy-assignment operator for the derived class (12.8), the *using-declaration* does not by itself suppress the implicit declaration of the derived class copy-assignment operator; the copy-assignment operator from the base class is hidden or overridden by the implicitly-declared copy-assignment operator of the derived class, as described below.

5 A *using-declaration* shall not name a *template-id*. [*Example:*

```
struct A {
  template <class T> void f(T);
  template <class T> struct X { };
};
struct B : A {
  using A::f<double>;           // ill-formed
  using A::X<int>;              // ill-formed
};
```

— *end example*]

6 A *using-declaration* shall not name a namespace.

7 A *using-declaration* shall not name a scoped enumerator.

8 A *using-declaration* for a class member shall be a *member-declaration*. [*Example:*

```
struct X {
  int i;
  static int s;
};

void f() {
```

```
  using X::i;       // error: X::i is a class member
                    // and this is not a member declaration.
  using X::s;       // error: X::s is a class member
                    // and this is not a member declaration.
}
```

— *end example*]

9 Members declared by a *using-declaration* can be referred to by explicit qualification just like other member names (3.4.3.2). In a *using-declaration*, a prefix `::` refers to the global namespace. [*Example:*

```
void f();

namespace A {
  void g();
}

namespace X {
  using ::f;        // global f
  using A::g;       // A's g
}

void h()
{
  X::f();           // calls ::f
  X::g();           // calls A::g
}
```

— *end example*]

10 A *using-declaration* is a *declaration* and can therefore be used repeatedly where (and only where) multiple declarations are allowed. [*Example:*

```
namespace A {
  int i;
}

namespace A1 {
  using A::i;
  using A::i;       // OK: double declaration
}

void f() {
  using A::i;
  using A::i;       // error: double declaration
}

struct B {
  int i;
};

struct X : B {
  using B::i;
  using B::i;       // error: double member declaration
};
```

— *end example*]

11 The entity declared by a *using-declaration* shall be known in the context using it according to its definition at the point of the *using-declaration*. Definitions added to the namespace after the *using-declaration* are not considered when a use of the name is made. [*Example:*

```
namespace A {
  void f(int);
}

using A::f;          // f is a synonym for A::f;
                     // that is, for A::f(int).
namespace A {
  void f(char);
}

void foo() {
  f('a');            // calls f(int),
}                    // even though f(char) exists.

void bar() {
  using A::f;        // f is a synonym for A::f;
                     // that is, for A::f(int) and A::f(char).
  f('a');            // calls f(char)
}
```

— *end example*]

12 [*Note:* partial specializations of class templates are found by looking up the primary class template and then considering all partial specializations of that template. If a *using-declaration* names a class template, partial specializations introduced after the *using-declaration* are effectively visible because the primary template is visible (14.5.5). — *end note*]

13 Since a *using-declaration* is a declaration, the restrictions on declarations of the same name in the same declarative region (3.3) also apply to *using-declaration*s. [*Example:*

```
namespace A {
  int x;
}

namespace B {
  int i;
  struct g { };
  struct x { };
  void f(int);
  void f(double);
  void g(char);      // OK: hides struct g
}

void func() {
  int i;
  using B::i;        // error: i declared twice
  void f(char);
  using B::f;        // OK: each f is a function
  f(3.5);            // calls B::f(double)
  using B::g;
  g('a');            // calls B::g(char)
```

```
  struct g g1;      // g1 has class type B::g
  using B::x;
  using A::x;       // OK: hides struct B::x
  x = 99;           // assigns to A::x
  struct x x1;      // x1 has class type B::x
}
```

— *end example*]

14 If a function declaration in namespace scope or block scope has the same name and the same parameter types as a function introduced by a *using-declaration*, and the declarations do not declare the same function, the program is ill-formed. [*Note:* two *using-declaration*s may introduce functions with the same name and the same parameter types. If, for a call to an unqualified function name, function overload resolution selects the functions introduced by such *using-declaration*s, the function call is ill-formed. [*Example:*

```
namespace B {
  void f(int);
  void f(double);
}
namespace C {
  void f(int);
  void f(double);
  void f(char);
}

void h() {
  using B::f;       // B::f(int) and B::f(double)
  using C::f;       // C::f(int), C::f(double), and C::f(char)
  f('h');           // calls C::f(char)
  f(1);             // error: ambiguous: B::f(int) or C::f(int)?
  void f(int);      // error: f(int) conflicts with C::f(int) and B::f(int)
}
```

— *end example*] — *end note*]

15 When a *using-declaration* brings names from a base class into a derived class scope, member functions and member function templates in the derived class override and/or hide member functions and member function templates with the same name, parameter-type-list (8.3.5), and cv-qualification in a base class (rather than conflicting). [*Note:* For *using-declarations* that name a constructor, see 12.9. — *end note*] [*Example:*

```
struct B {
  virtual void f(int);
  virtual void f(char);
  void g(int);
  void h(int);
};

struct D : B {
  using B::f;
  void f(int);      // OK: D::f(int) overrides B::f(int);

  using B::g;
  void g(char);     // OK

  using B::h;
  void h(int);      // OK: D::h(int) hides B::h(int)
```

```
};

void k(D* p)
{
  p->f(1);          // calls D::f(int)
  p->f('a');        // calls B::f(char)
  p->g(1);          // calls B::g(int)
  p->g('a');        // calls D::g(char)
}
```

— *end example*]

16 For the purpose of overload resolution, the functions which are introduced by a *using-declaration* into a derived class will be treated as though they were members of the derived class. In particular, the implicit `this` parameter shall be treated as if it were a pointer to the derived class rather than to the base class. This has no effect on the type of the function, and in all other respects the function remains a member of the base class.

17 The access rules for inheriting constructors are specified in 12.9; otherwise all instances of the name mentioned in a *using-declaration* shall be accessible. In particular, if a derived class uses a *using-declaration* to access a member of a base class, the member name shall be accessible. If the name is that of an overloaded member function, then all functions named shall be accessible. The base class members mentioned by a *using-declaration* shall be visible in the scope of at least one of the direct base classes of the class where the *using-declaration* is specified. [*Note:* because a *using-declaration* designates a base class member (and not a member subobject or a member function of a base class subobject), a *using-declaration* cannot be used to resolve inherited member ambiguities. For example,

```
struct A { int x(); };
struct B : A { };
struct C : A {
  using A::x;
  int x(int);
};

struct D : B, C {
  using C::x;
  int x(double);
};
int f(D* d) {
  return d->x();     // ambiguous: B::x or C::x
}
```

— *end note*]

18 The alias created by the *using-declaration* has the usual accessibility for a *member-declaration*. [*Note:* A *using-declaration* that names a constructor does not create aliases; see 12.9 for the pertinent accessibility rules. — *end note*] [*Example:*

```
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};
```

```
class B : public A {
  using A::f;       // error: A::f(char) is inaccessible
public:
  using A::g;       // B::g is a public synonym for A::g
};
```

— *end example*]

19 [*Note:* use of *access-declarations* (11.3) is deprecated; member *using-declarations* provide a better alternative. — *end note*]

20 If a *using-declaration* uses the keyword `typename` and specifies a dependent name (14.6.2), the name introduced by the *using-declaration* is treated as a *typedef-name* (7.1.3).

21 A *using-declaration* for a concept map is an alias to the concept map determined by concept map lookup (14.10.1.1) of the *concept-id* from the specified namespace. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
}
namespace N2 {
  concept_map N1::C<int> { } // A
  template<typename T> concept_map N1::C<T*> { } // B
}
namespace N3 {
  using N2::concept_map N1::C<int>;  // aliases A
  using N2::concept_map N1::C<int*>; // aliases B, instantiated with T=int
}
```

— *end example*]

22 A *using-declaration* for a concept map that specifies a *concept-name* (and not a *concept-id*) brings all of the concept maps and concept map templates from the specified namespace for the given concept into the scope in which the *using-declaration* appears. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
  template<C T> void f(T) { }
}
namespace N2 {
  concept_map N1::C<int> { } // A
  template<typename T> concept_map N1::C<T*> { } // B
}
namespace N3 {
  using N2::concept_map N1::C; // aliases A and B
  void g() {
    f(1); // uses concept map N1::C<int> from A
    f(new int); // uses concept map N1::C<int*> instantiated from B with T=int
  }
}
```

— *end example*]

23 If no concept is specified in the concept map *using-declaration*, the following rule applies. Let `X` be the namespace specified in the *nested-name-specifier* of the *using-declaration*. Let `S` be the set of all names of concept maps and concept map templates in `X` and in the transitive closure of all namespaces nominated

by *using-directive*s in X and its used namespaces, except that *using-directives* that nominate non-inline namespaces (7.3.1.2) are ignored in any namespace, including X, directly containing one or more names of a concept map or concept map template. No namespace is considered more than once to see if it contains a concept map or concept map template. If S is the empty set, the program is ill-formed. For each element in S, a name is introduced into the declarative region in which the *using-declaration* appears. The name is a synonym for the referent concept map or concept map template.

[*Example:*

```
namespace N1 {
  concept C<typename T> { }
  template<C T> void f(T) { }
}
namespace N2 {
  concept D<typename T> { }
}
namespace N3 {
  concept_map N1::C<int> { } // A
  template<typename T> concept_map N1::C<T*> { } // B
  concept_map N2::D<int> { } // C
}
namespace N4 {
  using N3::concept_map; // aliases A, B, and C
}
```

— *end example*]

24 If the second *nested-name-specifier* is specified but no concept is specified, then all concept maps in the namespace specified by the first *nested-name-specifier* for all concepts in the namespace specified by the second *nested-name-specifier* are brought into scope.

25 [*Note:* a *using-directive* for a namespace brings the concept maps of that namespace into scope, just like other entities. — *end note*] [*Example:*

```
namespace N1 {
  concept C<typename T> { }
}
namespace N2 {
  concept_map N1::C<int> { }
}
namespace N3 {
  using namespace N2;

  template<N1::C T> void foo(T) { };

  void bar() {
    foo(17); // OK, finds the concept map from N2
  }
}
```

— *end example*]

7.3.4 Using directive [namespace.udir]

using-directive:
attribute-specifier$_{opt}$ `using namespace` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name* `;`

1 A *using-directive* shall not appear in class scope, but may appear in namespace scope or in block scope. [*Note:* when looking up a *namespace-name* in a *using-directive*, only namespace names are considered, see 3.4.6. — *end note*] The optional *attribute-specifier* appertains to the *using-directive*.

2 A *using-directive* specifies that the names in the nominated namespace can be used in the scope in which the *using-directive* appears after the *using-directive*. During unqualified name lookup (3.4.1), the names appear as if they were declared in the nearest enclosing namespace which contains both the *using-directive* and the nominated namespace. [*Note:* in this context, "contains" means "contains directly or indirectly". — *end note*]

3 A *using-directive* does not add any members to the declarative region in which it appears. [*Example:*

```
namespace A {
  int i;
  namespace B {
    namespace C {
      int i;
    }
    using namespace A::B::C;
    void f1() {
      i = 5;          // OK, C::i visible in B and hides A::i
    }
  }
  namespace D {
    using namespace B;
    using namespace C;
    void f2() {
      i = 5;          // ambiguous, B::C::i or A::i?
    }
  }
  void f3() {
    i = 5;            // uses A::i
  }
}
void f4() {
  i = 5;              // ill-formed; neither i is visible
}
```

— *end example*]

4 The *using-directive* is transitive: if a scope contains a *using-directive* that nominates a second namespace that itself contains *using-directive*s, the effect is as if the *using-directive*s from the second namespace also appeared in the first. [*Example:*

```
namespace M {
  int i;
}

namespace N {
  int i;
  using namespace M;
}

void f() {
  using namespace N;
  i = 7;              // error: both M::i and N::i are visible
}
```

For another example,

```
namespace A {
  int i;
}
namespace B {
  int i;
  int j;
  namespace C {
    namespace D {
      using namespace A;
      int j;
      int k;
      int a = i;    // B::i hides A::i
    }
    using namespace D;
    int k = 89;     // no problem yet
    int l = k;      // ambiguous: C::k or D::k
    int m = i;      // B::i hides A::i
    int n = j;      // D::j hides B::j
  }
}
```

— *end example*]

5 If a namespace is extended by an *extension-namespace-definition* after a *using-directive* for that namespace is given, the additional members of the extended namespace and the members of namespaces nominated by *using-directive*s in the *extension-namespace-definition* can be used after the *extension-namespace-definition*.

6 If name lookup finds a declaration for a name in two different namespaces, and the declarations do not declare the same entity and do not declare functions, the use of the name is ill-formed. [*Note:* in particular, the name of an object, function or enumerator does not hide the name of a class or enumeration declared in a different namespace. For example,

```
namespace A {
  class X { };
  extern "C"   int g();
  extern "C++" int h();
}
namespace B {
  void X(int);
  extern "C"   int g();
  extern "C++" int h();
}
using namespace A;
using namespace B;

void f() {
  X(1);             // error: name X found in two namespaces
  g();              // okay: name g refers to the same entity
  h();              // error: name h found in two namespaces
}
```

— *end note*]

7 During overload resolution, all functions from the transitive search are considered for argument matching. The set of declarations found by the transitive search is unordered. [*Note:* in particular, the order in which

namespaces were considered and the relationships among the namespaces implied by the *using-directive*s do not cause preference to be given to any of the declarations found by the search. — *end note*] An ambiguity exists if the best match finds two functions with the same signature, even if one is in a namespace reachable through *using-directive*s in the namespace of the other.[86] [*Example:*

```
namespace D {
  int d1;
  void f(char);
}
using namespace D;

int d1;             // OK: no conflict with D::d1

namespace E {
  int e;
  void f(int);
}

namespace D {       // namespace extension
  int d2;
  using namespace E;
  void f(int);
}

void f() {
  d1++;             // error: ambiguous ::d1 or D::d1?
  ::d1++;           // OK
  D::d1++;          // OK
  d2++;             // OK: D::d2
  e++;              // OK: E::e
  f(1);             // error: ambiguous: D::f(int) or E::f(int)?
  f('a');           // OK: D::f(char)
}
```

— *end example*]

7.4 The asm declaration [dcl.asm]

1 An `asm` declaration has the form

asm-definition:
`asm` `(` *string-literal* `)` `;`

The `asm` declaration is conditionally-supported; its meaning is implementation-defined. [*Note:* Typically it is used to pass information through the implementation to an assembler. — *end note*]

7.5 Linkage specifications [dcl.link]

1 All function types, function names with external linkage, and variable names with external linkage have a *language linkage*. [*Note:* Some of the properties associated with an entity with language linkage are specific to each implementation and are not described here. For example, a particular language linkage may be associated with a particular form of representing names of objects and functions with external linkage, or with a particular calling convention, etc. — *end note*] The default language linkage of all function types,

86) During name lookup in a class hierarchy, some ambiguities may be resolved by considering whether one member hides the other along some paths (10.2). There is no such disambiguation when considering the set of names found as a result of following *using-directive*s.

function names, and variable names is C++ language linkage. Two function types with different language linkages are distinct types even if they are otherwise identical.

2 Linkage (3.5) between C++ and non-C++ code fragments can be achieved using a *linkage-specification*:

linkage-specification:
`extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
`extern` *string-literal declaration*

The *string-literal* indicates the required language linkage. This International Standard specifies the semantics for the *string-literal*s `"C"` and `"C++"`. Use of a *string-literal* other than `"C"` or `"C++"` is conditionally-supported, with implementation-defined semantics. [*Note:* Therefore, a linkage-specification with a *string-literal* that is unknown to the implementation requires a diagnostic. — *end note*] [*Note:* It is recommended that the spelling of the *string-literal* be taken from the document defining that language. For example, `Ada` (not `ADA`) and `Fortran` or `FORTRAN`, depending on the vintage. — *end note*]

3 Every implementation shall provide for linkage to functions written in the C programming language, `"C"`, and linkage to C++ functions, `"C++"`. [*Example:*

```
complex sqrt(complex);          // C++ linkage by default
extern "C" {
  double sqrt(double);          // C linkage
}
```

— *end example*]

4 Linkage specifications nest. When linkage specifications nest, the innermost one determines the language linkage. A linkage specification does not establish a scope. A *linkage-specification* shall occur only in namespace scope (3.3). In a *linkage-specification*, the specified language linkage applies to the function types of all function declarators, function names with external linkage, and variable names with external linkage declared within the *linkage-specification*. [*Example:*

```
extern "C" void f1(void(*pf)(int));
                                // the name f1 and its function type have C language
                                // linkage; pf is a pointer to a C function
extern "C" typedef void FUNC();
FUNC f2;                        // the name f2 has C++ language linkage and the
                                // function's type has C language linkage
extern "C" FUNC f3;             // the name of function f3 and the function's type
                                // have C language linkage
void (*pf2)(FUNC*);             // the name of the variable pf2 has C++ linkage and
                                // the type of pf2 is pointer to C++ function that
                                // takes one parameter of type pointer to C function
extern "C" {
  static void f4();             // the name of the function f4 has
                                // internal linkage (not C language
                                // linkage) and the function's type
                                // has C language linkage.
}

extern "C" void f5() {
  extern void f4();             // OK: Name linkage (internal)
                                // and function type linkage (C
                                // language linkage) gotten from
                                // previous declaration.
}
```

```
extern void f4();               // OK: Name linkage (internal)
                                // and function type linkage (C
                                // language linkage) gotten from
                                // previous declaration.
}

void f6() {
  extern void f4();             // OK: Name linkage (internal)
                                // and function type linkage (C
                                // language linkage) gotten from
                                // previous declaration.
}
```

— *end example*] A C language linkage is ignored for the names of class members and the member function type of class member functions. [*Example:*

```
extern "C" typedef void FUNC_c();
class C {
   void mf1(FUNC_c*);           // the name of the function mf1 and the member
                                // function's type have C++ language linkage; the
                                // parameter has type pointer to C function
   FUNC_c mf2;                  // the name of the function mf2 and the member
                                // function's type have C++ language linkage
   static FUNC_c* q;            // the name of the data member q has C++ language
                                // linkage and the data member's type is pointer to
                                // C function
};

extern "C" {
  class X {
  void mf();                    // the name of the function mf and the member
                                // function's type have C++ language linkage
  void mf2(void(*)());          // the name of the function mf2 has C++ language
                                // linkage; the parameter has type pointer to
                                // C function
  };
}
```

— *end example*]

5 If two declarations of the same function or object specify different *linkage-specification*s (that is, the *linkage-specification*s of these declarations specify different *string-literal*s), the program is ill-formed if the declarations appear in the same translation unit, and the one definition rule (3.2) applies if the declarations appear in different translation units. Except for functions with C++ linkage, a function declaration without a linkage specification shall not precede the first linkage specification for that function. A function can be declared without a linkage specification after an explicit linkage specification has been seen; the linkage explicitly specified in the earlier declaration is not affected by such a function declaration.

6 At most one function with a particular name can have C language linkage. Two declarations for a function with C language linkage with the same function name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same function. Two declarations for an object with C language linkage with the same name (ignoring the namespace names that qualify it) that appear in different namespace scopes refer to the same object. [*Note:* because of the one definition rule (3.2), only one definition for a function or object with C linkage may appear in the program; that is, such a function or object must not be defined in more than one namespace scope. For example,

```
namespace A {
  extern "C" int f();
  extern "C" int g() { return 1; }
  extern "C" int h();
}

namespace B {
  extern "C" int f();               // A::f and B::f refer
                                    // to the same function
  extern "C" int g() { return 1; }  // ill-formed, the function g
                                    // with C language linkage
                                    // has two definitions
}

int A::f() { return 98; }           //definition for the function f
                                    // with C language linkage
extern "C" int h() { return 97; }
                                    // definition for the function h
                                    // with C language linkage
                                    // A::h and ::h refer to the same function
```

— *end note*]

7 A declaration directly contained in a *linkage-specification* is treated as if it contains the `extern` specifier (7.1.1) for the purpose of determining the linkage of the declared name and whether it is a definition. Such a declaration shall not specify a storage class. [*Example:*

```
extern "C" double f();
static double f();                  // error
extern "C" int i;                   // declaration
extern "C" {
  int i;                            // definition
}
extern "C" static void g();         // error
```

— *end example*]

8 [*Note:* because the language linkage is part of a function type, when a pointer to C function (for example) is dereferenced, the function to which it refers is considered a C function. — *end note*]

9 Linkage from C++ to objects defined in other languages and to objects defined in C++ from other languages is implementation-defined and language-dependent. Only where the object layout strategies of two language implementations are similar enough can such linkage be achieved.

7.6 Attributes [dcl.attr]

7.6.1 Attribute syntax and semantics [dcl.attr.grammar]

1 Attributes specify additional information for various source constructs such as types, variables, names, blocks, or translation units.

attribute-specifier:
`[ [` *attribute-list* `] ]`

attribute-list:
attribute$_{opt}$
attribute-list `,` *attribute*$_{opt}$

attribute:
attribute-token attribute-argument-clause$_{opt}$

attribute-token:
identifier
attribute-scoped-token

attribute-scoped-token:
attribute-namespace `::` *identifier*

attribute-namespace:
identifier

attribute-argument-clause:
`(` *balanced-token-seq* `)`

balanced-token-seq:
balanced-token
balanced-token-seq balanced-token

balanced-token:
`(` *balanced-token-seq* `)`
`[` *balanced-token-seq* `]`
`{` *balanced-token-seq* `}`
any *token* other than a parenthesis, a bracket, or a brace

2 [*Note:* For each individual attribute, the form of the *balanced-token-seq* will be specified. — *end note*]

3 An *attribute-specifier* that contains no *attribute*s has no effect. The order in which the *attribute-tokens* appear in an *attribute-list* is not significant. A keyword (2.11) contained in an *attribute-token* is considered an identifier. No name lookup (3.4) is performed on any of the identifiers contained in an *attribute-token*. The *attribute-token* determines additional requirements on the *attribute-argument-clause* (if any). The use of an *attribute-scoped-token* is conditionally-supported, with implementation-defined behavior. [*Note:* Each implementation should choose a distinctive name for the *attribute-namespace* in an *attribute-scoped-token*. — *end note*]

4 Each *attribute-specifier* is said to *appertain* to some entity or statement, identified by the syntactic context where it appears (clause 7, clause 8). If an *attribute-specifier* that appertains to some entity or statement contains an *attribute* that does not apply to that entity or statement, the program is ill-formed. If an *attribute-specifier* appertains to a friend declaration (11.4), that declaration shall be a definition. No *attribute-specifier* shall appertain to an explicit instantiation (14.7.2).

5 For an *attribute-token* not specified in this International Standard, the behavior is implementation-defined.

7.6.2 Alignment attribute [dcl.align]

1 The *attribute-token* `align` specifies alignment. The *attribute* shall have one of the following forms:

```
align ( type-id )
align ( assignment-expression )
```

The attribute can be applied to a variable that is neither a function parameter nor declared with the register storage class specifier and to a class data member that is not a bit-field.

2 When the alignment attribute is of the form `align(`*assignment-expression* `)`:

— the *assignment-expression* shall be an integral constant expression

— if the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared object shall be the specified fundamental alignment

— if the constant expression evaluates to an extended alignment and the implementation supports that alignment in the context of the declaration, the alignment of the declared object shall be that alignment

— if the constant expression evaluates to an extended alignment and the implementation does not support that alignment in the context of the declaration, the program is ill-formed

— if the constant expression evaluates to zero, the alignment specifier shall have no effect

— otherwise, the program is ill-formed.

3 When the alignment attribute is of the form `align(` *type-id* `)`, it shall have the same effect as `align(alignof(` *type-id* `))` (5.3.6).

4 When multiple alignment attributes are specified for an object, the alignment requirement shall be set to the strictest specified alignment.

5 The combined effect of all alignment attributes in a declaration shall not specify an alignment that is less strict than the alignment that would otherwise be required for the object being declared.

6 If the defining declaration of an object has an alignment attribute, any non-defining declaration of that object shall either specify equivalent alignment or have no alignment attribute. No diagnostic is required if declarations of an object have different alignment attributes in different translation units.

7 [*Example:* An aligned buffer with an alignment requirement of `A` and holding `N` elements of type `T` other than `char`, `signed char`, or `unsigned char` can be declared as:

```
T buffer [[ align(T), align(A) ]] [N];
```

Specifying `align(T)` in the *attribute-list* ensures that the final requested algnment will not be weaker than `alignof(T)`, and therefore the program will not be ill-formed. — *end example*]

8 [*Note:* the alignment of a union type can be strengthened by applying the alignment attribute to any member of the union. — *end note*]

9 [*Example:*

```
void f [[ align(double) ]] ();            // error: alignment applied to function
unsigned char c
  [[ align(double) ]] [sizeof(double)];   // array of characters, suitably aligned for a double
extern unsigned char c[sizeof(double)];   // no align necessary
extern unsigned char c
  [[ align(float) ]] [sizeof(double)];    // error: different alignment in declaration
```

— *end example*]

7.6.3 Noreturn attribute [dcl.attr.noreturn]

1 The *attribute-token* `noreturn` specifies that a function does not return. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to the *declarator-id* in a function declaration. The first declaration of a function shall specify the `noreturn` attribute if any declaration of that function specifies the `noreturn` attribute. If a function is declared with the `noreturn` attribute in one translation unit and the same function is declared without the `noreturn` attribute in another translation unit, the program is ill-formed; no diagnostic required.

2 If a function `f` is called where `f` was previously declared with the `noreturn` attribute and `f` eventually returns, the behavior is undefined. [*Note:* The function may terminate by throwing an exception. — *end note*]

3 [*Example:*

```
void f [[ noreturn ]] () {
  throw "error";        // OK
}

void q [[ noreturn ]] (int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
}
```

— *end example*]

7.6.4 Final attribute [dcl.attr.final]

1 The *attribute-token* `final` specifies overriding semantics for a virtual function. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to class definitions and to virtual member functions being declared in a class definition. If the attribute is specified for a class definition, it is equivalent to being specified for each virtual member function of that class, including inherited member functions.

2 If a virtual member function `f` in some class `B` is marked `final` and in a class `D` derived from `B` a function `D::f` overrides `B::f`, the program is ill-formed; no diagnostic required.[87]

3 [*Example:*

```
struct B {
  virtual void f [[ final ]] ();
};

struct D : B {
  void f();     // ill-formed
};
```

— *end example*]

7.6.5 Carries dependency attribute [dcl.attr.depend]

1 The *attribute-token* `carries_dependency` specifies dependency propagation into and out of functions. It shall appear at most once in each *attribute-list* and no *attribute-argument-clause* shall be present. The attribute applies to the *declarator-id* of a *parameter-declaration*, in which case it specifies that the initialization of the parameter carries a dependency to (1.10) each lvalue-to-rvalue conversion (4.1) of that object. The attribute also applies to the *declarator-id* of a function declaration, in which case it specifies that the return value carries a dependency to the evaluation of the function call expression.

2 The first declaration of a function shall specify the `carries_dependency` attribute for its *declarator-id* if any declaration of the function specifies the `carries_dependency` attribute. Furthermore, the first declaration of a function shall specify the `carries_dependency` attribute for a parameter if any declaration of that function specifies the `carries_dependency` attribute for that parameter. If a function or one of its parameters is declared with the `carries_dependency` attribute in its first declaration in one translation unit and the same function or one of its parameters is declared without the `carries_dependency` attribute in its first declaration in another translation unit, the program is ill-formed; no diagnostic required.

3 [*Note:* the `carries_dependency` attribute does not change the meaning of the program, but may result in generation of more efficient code. — *end note*]

4 [*Example:*

87) If an implementation does not emit a diagnostic it should execute the program as if `final` were not present.

```
/* Compilation unit A. */

struct foo { int* a; int* b; };
struct foo* foo_head[10];

struct foo* f [[carries_dependency]] (int i) {
  return foo_head[i].load(memory_order_consume);
}

int g(int* x, int* y [[carries_dependency]]) {
  return kill_dependency(foo_array[*x][*y]);
}

/* Compilation unit B. */

struct foo* f [[carries_dependency]] (int i);
int* g(int* x, int* y [[carries_dependency]]);

int c = 3;

void h(int i) {
  struct foo* p;

  p = f(i);
  do_something_with(g(&c, p->a));
  do_something_with(g(p->a, &c));
}
```

5 The annotation on function `f` means that the return value carries a dependency out of `f`, so that the implementation need not constrain ordering upon return from `f`.

6 Function `g`'s second argument is annotated, but its first argument is not. Therefore, function `h`'s first call to `g` carries a dependency into `g`, but its second call does not. The implementation might need to constrain ordering prior to the second call to `g`.

— *end example*]

8 Declarators [dcl.decl]

1 A declarator declares a single object, function, or type, within a declaration. The *init-declarator-list* appearing in a declaration is a comma-separated sequence of declarators, each of which can have an initializer.

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*

init-declarator:
declarator initializer$_{opt}$

2 The two components of a *declaration* are the specifiers (*decl-specifier-seq*; 7.1) and the declarators (*init-declarator-list*). The specifiers indicate the type, storage class or other properties of the objects, functions or typedefs being declared. The declarators specify the names of these objects, functions or typedefs, and (optionally) modify the type of the specifiers with operators such as `*` (pointer to) and `()` (function returning). Initial values can also be specified in a declarator; initializers are discussed in 8.5 and 12.6.

3 Each *init-declarator* in a declaration is analyzed separately as if it was in a declaration by itself.[88]

4 Declarators have the syntax

declarator:
ptr-declarator
noptr-declarator parameters-and-qualifiers `->` *attribute-specifier*$_{opt}$ *type-id*

ptr-declarator:
noptr-declarator
ptr-operator ptr-declarator

noptr-declarator:
declarator-id attribute-specifier$_{opt}$
noptr-declarator parameters-and-qualifiers
noptr-declarator `[` *constant-expression*$_{opt}$ `]` *attribute-specifier*$_{opt}$
`(` *ptr-declarator* `)`

parameters-and-qualifiers:
`(` *parameter-declaration-clause* `)` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$

88) A declaration with several declarators is usually equivalent to the corresponding sequence of declarations each with a single declarator. That is
```
T D1, D2, ... Dn;
```
is usually equvalent to
```
T D1; T D2; ... T Dn;
```
where `T` is a *decl-specifier-seq* and each `Di` is an *init-declarator*. The exception occurs when a name introduced by one of the *declarators* hides a type name used by the *dcl-specifiers*, so that when the same *dcl-specifiers* are used in a subsequent declaration, they do not have the same meaning, as in
```
struct S ... ;
S S, T; // declare two instances of struct S
```
which is not equivalent to
```
struct S ... ;
S S;
S T; // error
```

ptr-operator:
`*` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
`&`
`&&`
`::`$_{opt}$ *nested-name-specifier* `*` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
`const`
`volatile`

ref-qualifier:
`&`
`&&`

declarator-id:
`...`$_{opt}$ *id-expression*
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

A *class-name* has special meaning in a declaration of the class of that name and when qualified by that name using the scope resolution operator `::` (5.1, 12.1, 12.4).

8.1 Type names [dcl.name]

1 To specify type conversions explicitly, and as an argument of `sizeof`, `alignof`, `new`, or `typeid`, the name of a type shall be specified. This can be done with a *type-id*, which is syntactically a declaration for an object or function of that type that omits the name of the object or function.

type-id:
type-specifier-seq attribute-specifier$_{opt}$ *abstract-declarator*$_{opt}$

abstract-declarator:
ptr-abstract-declarator
noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers* `->` *attribute-specifier*$_{opt}$ *type-id*
`...`

ptr-abstract-declarator:
noptr-abstract-declarator
ptr-operator ptr-abstract-declarator$_{opt}$

noptr-abstract-declarator:
noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers*
noptr-abstract-declarator$_{opt}$ `[` *constant-expression* `]` *attribute-specifier*$_{opt}$
`(` *ptr-abstract-declarator* `)`

It is possible to identify uniquely the location in the *abstract-declarator* where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. [*Example:*

```
int                 // int i
int *               // int *pi
int *[3]            // int *p[3]
int (*)[3]          // int (*p3i)[3]
int *()             // int *f()
int (*)(double)     // int (*pf)(double)
```

name respectively the types "`int`," "pointer to `int`," "array of 3 pointers to `int`," "pointer to array of 3 `int`," "function of (no parameters) returning pointer to `int`," and "pointer to a function of (`double`) returning `int`." — *end example*]

2 A type can also be named (often more easily) by using a *typedef* (7.1.3).

8.2 Ambiguity resolution [dcl.ambig.res]

1 The ambiguity arising from the similarity between a function-style cast and a declaration mentioned in 6.8 can also occur in the context of a declaration. In that context, the choice is between a function declaration with a redundant set of parentheses around a parameter name and an object declaration with a function-style cast as the initializer. Just as for the ambiguities mentioned in 6.8, the resolution is to consider any construct that could possibly be a declaration a declaration. [*Note:* a declaration can be explicitly disambiguated by a nonfunction-style cast, by an `=` to indicate initialization or by removing the redundant parentheses around the parameter name. — *end note*] [*Example:*

```
struct S {
  S(int);
};

void foo(double a) {
  S w(int(a));       // function declaration
  S x(int());        // function declaration
  S y((int)a);       // object declaration
  S z = int(a);      // object declaration
}
```

— *end example*]

2 The ambiguity arising from the similarity between a function-style cast and a *type-id* can occur in different contexts. The ambiguity appears as a choice between a function-style cast expression and a declaration of a type. The resolution is that any construct that could possibly be a *type-id* in its syntactic context shall be considered a *type-id*.

3 [*Example:*

```
#include <cstddef>
char *p;
void *operator new(std::size_t, int);
void foo()  {
  const int x = 63;
  new (int(*p)) int;            // new-placement expression
  new (int(*[x]));              // new type-id
}
```

4 For another example,

```
template <class T>
struct S {
  T *p;
};
S<int()> x;                     // type-id
S<int(1)> y;                    // expression (ill-formed)
```

5 For another example,

```
void foo() {
  sizeof(int(1));               // expression
  sizeof(int());                // type-id (ill-formed)
}
```

6 For another example,

```
void foo() {
  (int(1));                    // expression
  (int())1;                    // type-id (ill-formed)
}
```

— *end example*]

7 Another ambiguity arises in a *parameter-declaration-clause* of a function declaration, or in a *type-id* that is the operand of a `sizeof` or `typeid` operator, when a *type-name* is nested in parentheses. In this case, the choice is between the declaration of a parameter of type pointer to function and the declaration of a parameter with redundant parentheses around the *declarator-id*. The resolution is to consider the *type-name* as a *simple-type-specifier* rather than a *declarator-id*. [*Example:*

```
class C { };
void f(int(C)) { }             // void f(int(*fp)(C c)) { }
                               // not: void f(int C);

int g(C);

void foo() {
  f(1);                        // error: cannot convert 1 to function pointer
  f(g);                        // OK
}
```

For another example,

```
class C { };
void h(int *(C[10]));          // void h(int *(*_fp)(C _parm[10]));
                               // not: void h(int *C[10]);
```

— *end example*]

8.3 Meaning of declarators [dcl.meaning]

1 A list of declarators appears after an optional (Clause 7) *decl-specifier-seq* (7.1). Each declarator contains exactly one *declarator-id*; it names the identifier that is declared. An *unqualified-id* occurring in a *declarator-id* shall be a simple *identifier* except for the declaration of some special functions (12.3, 12.4, 13.5) and for the declaration of template specializations or partial specializations (14.7). A *declarator-id* shall not be qualified except for the definition of a member function (9.3) or static data member (9.4) outside of its class, the definition or explicit instantiation of a function or variable member of a namespace outside of its namespace, or the definition of a previously declared explicit specialization outside of its namespace, or the declaration of a friend function that is a member of another class or namespace (11.4). When the *declarator-id* is qualified, the declaration shall refer to a previously declared member of the class or namespace to which the qualifier refers (or of an inline namespace within that scope (7.3.1)), and the member shall not have been introduced by a *using-declaration* in the scope of the class or namespace nominated by the *nested-name-specifier* of the *declarator-id*. [*Note:* if the qualifier is the global `::` scope resolution operator, the *declarator-id* refers to a name declared in the global namespace scope. — *end note*] The optional *attribute-specifier* following a *declarator-id* appertains to the entity that is declared.

2 A `static`, `thread_local`, `extern`, `register`, `mutable`, `friend`, `inline`, `virtual`, or `typedef` specifier applies directly to each *declarator-id* in an *init-declarator-list*; the type specified for each *declarator-id* depends on both the *decl-specifier-seq* and its *declarator*.

3 Thus, a declaration of a particular identifier has the form

```
T D
```

where `T` is of the form *attribute-specifier*$_{opt}$ *decl-specifier-seq* *attribute-specifier*$_{opt}$ and `D` is a declarator. Following is a recursive procedure for determining the type specified for the contained *declarator-id* by such a declaration.

4 First, the *decl-specifier-seq* determines a type. In a declaration

```
T D
```

the *decl-specifier-seq* `T` determines the type `T`. [*Example:* in the declaration

```
int unsigned i;
```

the type specifiers `int unsigned` determine the type "`unsigned int`" (7.1.6.2). — *end example*]

5 In a declaration *attribute-specifier*$_{opt}$ `T` *attribute-specifier*$_{opt}$ `D` where `D` is an unadorned identifier the type of this identifier is "`T`". The first optional *attribute-specifier* appertains to the entity being declared. The second optional *attribute-specifier* appertains to the type `T`, but not to the class or enumeration declared in the *decl-specifier-seq*, if any.

6 In a declaration `T D` where `D` has the form

(D1)

the type of the contained *declarator-id* is the same as that of the contained *declarator-id* in the declaration

```
T D1
```

Parentheses do not alter the type of the embedded *declarator-id*, but they can alter the binding of complex declarators.

7 In a constrained context (14.10), a type archetype *cv* `T` shall be used as the type of a variable only if the template has a concept requirement `std::VariableType<T>`.

8.3.1 Pointers [dcl.ptr]

1 In a declaration `T D` where `D` has the form

`*` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$ `D1`

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to `T`." The *cv-qualifier*s apply to the pointer and not to the object pointed to. Similarly, the optional *attribute-specifier* (7.6.1) appertains to the pointer and not to the object pointed to.

2 [*Example:* the declarations

```
const int ci = 10, *pc = &ci, *const cpc = pc, **ppc;
int i, *p, *const cp = &i;
```

declare `ci`, a constant integer; `pc`, a pointer to a constant integer; `cpc`, a constant pointer to a constant integer; `ppc`, a pointer to a pointer to a constant integer; `i`, an integer; `p`, a pointer to integer; and `cp`, a constant pointer to integer. The value of `ci`, `cpc`, and `cp` cannot be changed after initialization. The value of `pc` can be changed, and so can the object pointed to by `cp`. Examples of some correct operations are

```
i = ci;
*cp = ci;
pc++;
```

```
pc = cpc;
pc = p;
ppc = &pc;
```

Examples of ill-formed operations are

```
ci = 1;             // error
ci++;               // error
*pc = 2;            // error
cp = &ci;           // error
cpc++;              // error
p = pc;             // error
ppc = &p;           // error
```

Each is unacceptable because it would either change the value of an object declared `const` or allow it to be changed through a cv-unqualified pointer later, for example:

```
*ppc = &ci;         // OK, but would make p point to ci ...
                    // ... because of previous error
*p = 5;             // clobber ci
```

— *end example*]

3 See also 5.17 and 8.5.

4 [*Note:* there are no pointers to references; see 8.3.2. Since the address of a bit-field (9.6) cannot be taken, a pointer can never point to a bit-field. — *end note*]

5 In a constrained context (14.10), a type archetype *cv* `T` shall be used to form a type "pointer to *cv* `T`" only if the template has a concept requirement `std::PointeeType<T>`.

8.3.2 References [dcl.ref]

1 In a declaration `T D` where `D` has either of the forms

```
& D1
&& D1
```

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`," then the type of the identifier of `D` is "*derived-declarator-type-list* reference to `T`." Cv-qualified references are ill-formed except when the cv-qualifiers are introduced through the use of a typedef (7.1.3) or of a template type argument (14.3), in which case the cv-qualifiers are ignored. [*Example:*

```
typedef int& A;
const A aref = 3;   // ill-formed; non-const reference initialized with rvalue
```

The type of `aref` is "reference to `int`", not "`const` reference to `int`". — *end example*] [*Note:* a reference can be thought of as a name of an object. — *end note*] A declarator that specifies the type "reference to *cv* `void`" is ill-formed.

2 A reference type that is declared using `&` is called an *lvalue reference*, and a reference type that is declared using `&&` is called an *rvalue reference*. Lvalue references and rvalue references are distinct types. Except where explicitly noted, they are semantically equivalent and commonly referred to as references.

3 [*Example:*

```
void f(double& a) { a += 3.14; }
// ...
double d = 0;
f(d);
```

declares `a` to be a reference parameter of `f` so the call `f(d)` will add `3.14` to `d`.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;
```

declares the function `g()` to return a reference to an integer so `g(3)=7` will assign `7` to the fourth element of the array `v`. For another example,

```
struct link {
  link* next;
};

link* first;

void h(link*& p) {  // p is a reference to pointer
  p->next = first;
  first = p;
  p = 0;
}

void k() {
   link* q = new link;
   h(q);
}
```

declares `p` to be a reference to a pointer to `link` so `h(q)` will leave `q` with the value zero. See also 8.5.3. — *end example*]

4 It is unspecified whether or not a reference requires storage (3.7).

5 There shall be no references to references, no arrays of references, and no pointers to references. The declaration of a reference shall contain an *initializer* (8.5.3) except when the declaration contains an explicit `extern` specifier (7.1.1), is a class member (9.2) declaration within a class definition, or is the declaration of a parameter or a return type (8.3.5); see 3.1. A reference shall be initialized to refer to a valid object or function. [*Note:* in particular, a null reference cannot exist in a well-defined program, because the only way to create such a reference would be to bind it to the "object" obtained by dereferencing a null pointer, which causes undefined behavior. As described in 9.6, a reference cannot be bound directly to a bit-field. — *end note*]

6 In a constrained context (14.10), a type archetype *cv* `T` shall be used to form a type "reference to *cv* `T`" only if the template has a concept requirement `std::ReferentType<T>`.

8.3.3 Pointers to members [dcl.mptr]

1 In a declaration `T D` where `D` has the form

$$::_{opt} \quad \textit{nested-name-specifier} \texttt{ *} \ \textit{attribute-specifier}_{opt} \ \textit{cv-qualifier-seq}_{opt} \ \textit{D1}$$

and the *nested-name-specifier* names a class, and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is "*derived-declarator-type-list cv-qualifier-seq* pointer to member of class *nested-name-specifier* of type `T`". The optional *attribute-specifier* (7.6.1) appertains to the pointer-to-member.

2 [*Example:*

```
struct X {
  void f(int);
  int a;
};
struct Y;

int X::* pmi = &X::a;
void (X::* pmf)(int) = &X::f;
double X::* pmd;
char Y::* pmc;
```

declares `pmi`, `pmf`, `pmd` and `pmc` to be a pointer to a member of `X` of type `int`, a pointer to a member of `X` of type `void(int)`, a pointer to a member of `X` of type `double` and a pointer to a member of `Y` of type `char` respectively. The declaration of `pmd` is well-formed even though `X` has no members of type `double`. Similarly, the declaration of `pmc` is well-formed even though `Y` is an incomplete type. `pmi` and `pmf` can be used like this:

```
X obj;
// ...
obj.*pmi = 7;        // assign 7 to an integer
                     // member of obj
(obj.*pmf)(7);       // call a function member of obj
                     // with the argument 7
```

— *end example*]

3 A pointer to member shall not point to a static member of a class (9.4), a member with reference type, or "*cv* `void`." In a constrained context (14.10), a pointer to member shall point to a type archetype *cv* `T` only if the template has a concept requirement `std::MemberPointeeType<T>`.

[*Note:* see also 5.3 and 5.5. The type "pointer to member" is distinct from the type "pointer", that is, a pointer to member is declared only by the pointer to member declarator syntax, and never by the pointer declarator syntax. There is no "reference-to-member" type in C++. — *end note*]

8.3.4 Arrays [dcl.array]

1 In a declaration `T D` where `D` has the form

`D1 [` *constant-expression*$_{opt}$ `]` *attribute-specifier*$_{opt}$

and the type of the identifier in the declaration `T D1` is "*derived-declarator-type-list* `T`", then the type of the identifier of `D` is an array type; if the type of the identifier of `D` contains the `auto` *type-specifier*, the program is ill-formed. `T` is called the array *element type*; this type shall not be a reference type, the (possibly cv-qualified) type `void`, a function type or an abstract class type. If the *constant-expression* (5.19) is present, it shall be an integral constant expression and its value shall be greater than zero. The constant expression specifies the *bound* of (number of elements in) the array. If the value of the constant expression is `N`, the array has `N` elements numbered `0` to `N-1`, and the type of the identifier of `D` is "*derived-declarator-type-list* array of `N T`". An object of array type contains a contiguously allocated non-empty set of `N` subobjects of type `T`. If the constant expression is omitted, the type of the identifier of `D` is "*derived-declarator-type-list* array

of unknown bound of `T`", an incomplete effective object type. The type "*derived-declarator-type-list* array of `N T`" is a different type from the type "*derived-declarator-type-list* array of unknown bound of `T`", see 3.9. Any type of the form "*cv-qualifier-seq* array of `N T`" is adjusted to "array of `N` *cv-qualifier-seq* `T`", and similarly for "array of unknown bound of `T`". The optional *attribute-specifier* appertains to the array. [*Example:*

```
typedef int A[5], AA[2][3];
typedef const A CA;             // type is "array of 5 const int"
typedef const AA CAA;           // type is "array of 2 array of 3 const int"
```

— *end example*] [*Note:* an "array of `N` *cv-qualifier-seq* `T`" has cv-qualified type; see 3.9.3. — *end note*]

2 An array can be constructed from one of the fundamental types (except `void`), from a pointer, from a pointer to member, from a class, from an enumeration type, or from another array. In a constrained context (14.10), an array shall be constructed from a type archetype *cv* `T` only if the template has a concept requirement `std::ValueType<T>`.

3 When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions that specify the bounds of the arrays can be omitted only for the first member of the sequence. [*Note:* this elision is useful for function parameters of array types, and when the array is external and the definition, which allocates storage, is given elsewhere. — *end note*] The first *constant-expression* can also be omitted when the declarator is followed by an *initializer* (8.5). In this case the bound is calculated from the number of initial elements (say, `N`) supplied (8.5.1), and the type of the identifier of `D` is "array of `N T`."

4 [*Example:*

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. For another example,

```
static int x3d[3][5][7];
```

declares a static three-dimensional array of integers, with rank $3 \times 5 \times 7$. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` can reasonably appear in an expression. — *end example*]

5 [*Note:* conversions affecting lvalues of array type are described in 4.2. Objects of array types cannot be modified, see 3.10. — *end note*]

6 Except where it has been declared for a class (13.5.5), the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+(E2))`. Because of the conversion rules that apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2`-th member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

7 A consistent rule is followed for multidimensional arrays. If `E` is an n-dimensional array of rank $i \times j \times \ldots \times k$, then `E` appearing in an expression is converted to a pointer to an $(n-1)$-dimensional array with rank $j \times \ldots \times k$. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to $(n-1)$-dimensional array, which itself is immediately converted into a pointer.

8 [*Example:* consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) five-membered arrays of integers. In the expression `x[i]` which is equivalent to `*(x+i)`, `x` is first converted to a pointer as described; then `x+i` is converted to the type of `x`, which involves multiplying `i` by the length of the object to which the pointer points, namely five integer objects. The results are added

and indirection applied to yield an array (of five integers), which in turn is converted to a pointer to the first of the integers. If there is another subscript the same argument applies again; this time the result is an integer. — *end example*]

9 [*Note:* it follows from all this that arrays in C++ are stored row-wise (last subscript varies fastest) and that the first subscript in the declaration helps determine the amount of storage consumed by an array but plays no other part in subscript calculations. — *end note*]

8.3.5 Functions [dcl.fct]

1 In a declaration `T D` where `D` has the form

`D1 (` *parameter-declaration-clause* `)` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", the type of the *declarator-id* in `D` is "*derived-declarator-type-list* function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning `T`". The optional *attribute-specifier* appertains to the function type.

2 In a declaration `T D` where `D` has the form

`D1 (` *parameter-declaration-clause* `)` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$ `->` *attribute-specifier*$_{opt}$ *type-id*

and the type of the contained *declarator-id* in the declaration `T D1` is "*derived-declarator-type-list* `T`", `T` shall be the single *type-specifier* `auto`. The type of the *declarator-id* in `D` is "function of (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ returning *type-id*". Such a function type has a *late-specified return type*. The first optional *attribute-specifier* appertains to the function type. The second optional *attribute-specifier* appertains to the return type.

3 The *type-id* in this form includes the longest possible sequence of *abstract-declarator*s. [*Note:* This resolves the ambiguous binding of array and function declarators. [*Example:*

```
auto f()->int(*)[4]; // function returning a pointer to array[4] of int
                     // not function returning array[4] of pointer to int
```

— *end example*] — *end note*]

4 A type of either form is a *function type*.[89]

parameter-declaration-clause:
parameter-declaration-list$_{opt}$ *...*$_{opt}$
parameter-declaration-list `,` *...*

parameter-declaration-list:
parameter-declaration
parameter-declaration-list `,` *parameter-declaration*

parameter-declaration:
decl-specifier-seq attribute-specifier$_{opt}$ *declarator*
decl-specifier-seq attribute-specifier$_{opt}$ *declarator* `=` *assignment-expression*
decl-specifier-seq attribute-specifier$_{opt}$ *abstract-declarator*$_{opt}$
decl-specifier-seq attribute-specifier$_{opt}$ *abstract-declarator*$_{opt}$ `=` *assignment-expression*

5 The *parameter-declaration-clause* determines the arguments that can be specified, and their processing, when the function is called. [*Note:* the *parameter-declaration-clause* is used to convert the arguments specified on the function call; see 5.2.2. — *end note*] If the *parameter-declaration-clause* is empty, the function takes no arguments. The parameter list `(void)` is equivalent to the empty parameter list. Except for this special case, `void` shall not be a parameter type (though types derived from `void`, such as `void*`, can).

89) As indicated by syntax, cv-qualifiers are a signficant component in function return types.

If the *parameter-declaration-clause* terminates with an ellipsis or a function parameter pack (14.5.3), the number of arguments shall be equal to or greater than the number of parameters that do not have a default argument and are not function parameter packs. Where syntactically correct and where "..." is not part of an *abstract-declarator*, ", ..." is synonymous with "...". [*Example:* the declaration

```
int printf(const char*, ...);
```

declares a function that can be called with varying numbers and types of arguments.

```
printf("hello world");
printf("a=%d b=%d", a, b);
```

However, the first argument must be of a type that can be converted to a `const char*` — *end example*] [*Note:* the standard header `<cstdarg>` contains a mechanism for accessing arguments passed using the ellipsis (see 5.2.2 and 18.9). — *end note*]

6 A single name can be used for several different functions in a single scope; this is function overloading (Clause 13). All declarations for a function shall agree exactly in both the return type and the parameter-type-list. The type of a function is determined using the following rules. The type of each parameter (including function parameter packs) is determined from its own *decl-specifier-seq* and *declarator*. After determining the type of each parameter, any parameter of type "array of `T`" or "function returning `T`" is adjusted to be "pointer to `T`" or "pointer to function returning `T`," respectively. After producing the list of parameter types, several transformations take place upon these types to determine the function type. Any *cv-qualifier* modifying a parameter type is deleted. [*Example:* the type `void(*)(const int)` becomes `void(*)(int)` — *end example*] Such *cv-qualifier*s affect only the definition of the parameter within the body of the function; they do not affect the function type. If a *storage-class-specifier* modifies a parameter type, the specifier is deleted. [*Example:* `register char*` becomes `char*` — *end example*] Such *storage-class-specifier*s affect only the definition of the parameter within the body of the function; they do not affect the function type. The resulting list of transformed parameter types and the presence or absence of the ellipsis or a function parameter pack is the function's *parameter-type-list*.

7 A *cv-qualifier-seq* shall only be part of the function type for a non-static member function, the function type to which a pointer to member refers, or the top-level function type of a function typedef declaration. The effect of a *cv-qualifier-seq* in a function declarator is not the same as adding cv-qualification on top of the function type. In the latter case, the cv-qualifiers are ignored. [*Example:*

```
typedef void F();
struct S {
  const F f;        // OK: equivalent to: void f();
};
```

— *end example*] A *ref-qualifier* shall only be part of the function type for a non-static member function, the function type to which a pointer to member refers, or the top-level function type of a function typedef declaration. The return type, the parameter-type-list, the *ref-qualifier*, and the *cv-qualifier-seq*, but not the default arguments (8.3.6) or the exception specification (15.4), are part of the function type. [*Note:* function types are checked during the assignments and initializations of pointer-to-functions, reference-to-functions, and pointer-to-member-functions. — *end note*]

8 [*Example:* the declaration

```
int fseek(FILE*, long, int);
```

declares a function taking three arguments of the specified types, and returning `int` (7.1.6). — *end example*]

9 If the type of a parameter includes a type of the form "pointer to array of unknown bound of `T`" or "reference to array of unknown bound of `T`," the program is ill-formed.[90] Functions shall not have a return type of type array or function, although they may have a return type of type pointer or reference to such things. There shall be no arrays of functions, although there can be arrays of pointers to functions. In a constrained context (14.10), a type archetype *cv* `T` shall be used as the return type of a funcion type if only the template has a concept requirement `std::Returnable<T>`.

Types shall not be defined in return or parameter types. The type of a parameter or the return type for a function definition shall not be an incomplete class type (possibly cv-qualified) unless the function definition is nested within the *member-specification* for that class (including definitions in nested classes defined within the class).

10 A typedef of function type may be used to declare a function but shall not be used to define a function (8.4). [*Example:*

```
typedef void F();
F  fv;              // OK: equivalent to void fv();
F  fv { }           // ill-formed
void fv() { }       // OK: definition of fv
```

— *end example*] A typedef of a function type whose declarator includes a *cv-qualifier-seq* shall be used only to declare the function type for a non-static member function, to declare the function type to which a pointer to member refers, or to declare the top-level function type of another function typedef declaration. [*Example:*

```
typedef int FIC(int) const;
FIC f;              // ill-formed: does not declare a member function
struct S {
  FIC f;            // OK
};
FIC S::*pm = &S::f; // OK
```

— *end example*]

11 An identifier can optionally be provided as a parameter name; if present in a function definition (8.4), it names a parameter (sometimes called "formal argument"). [*Note:* in particular, parameter names are also optional in function definitions and names used for a parameter in different declarations and the definition of a function need not be the same. If a parameter name is present in a function declaration that is not a definition, it cannot be used outside of the *parameter-declaration-clause* since it goes out of scope at the end of the function declarator (3.3). — *end note*]

12 [*Example:* the declaration

```
int i,
    *pi,
    f(),
    *fpi(int),
    (*pif)(const char*, const char*),
    (*fpif(int))(int);
```

declares an integer `i`, a pointer `pi` to an integer, a function `f` taking no arguments and returning an integer, a function `fpi` taking an integer argument and returning a pointer to an integer, a pointer `pif` to a function

90) This excludes parameters of type "*ptr-arr-seq* `T2`" where `T2` is "pointer to array of unknown bound of `T`" and where *ptr-arr-seq* means any sequence of "pointer to" and "array of" derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.

which takes two pointers to constant characters and returns an integer, a function `fpif` taking an integer argument and returning a pointer to a function that takes an integer argument and returns an integer. It is especially useful to compare `fpi` and `pif`. The binding of `*fpi(int)` is `*(fpi(int))`, so the declaration suggests, and the same construction in an expression requires, the calling of a function `fpi`, and then using indirection through the (pointer) result to yield an integer. In the declarator `(*pif)(const char*, const char*)`, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function, which is then called. — *end example*] [*Note:* typedefs and late-specified return types are sometimes convenient when the return type of a function is complex. For example, the function `fpif` above could have been declared

```
typedef int  IFUNC(int);
IFUNC*  fpif(int);
```

or

```
auto fpif(int)->int(*)(int)
```

A late-specified return type is most useful for a type that would be more complicated to specify before the *declarator-id*:

```
template <class T, class U> auto add(T t, U u) -> decltype(t + u);
```

rather than

```
template <class T, class U> decltype((*(T*)0) + (*(U*)0)) add(T t, U u);
```

— *end note*]

13 A *declarator-id* or *abstract-declarator* containing an ellipsis shall only be used in a *parameter-declaration*. Such a *parameter-declaration* is a parameter pack (14.5.3). When it is part of a *parameter-declaration-clause*, the parameter pack is a function parameter pack (14.5.3). [*Note:* Otherwise, the *parameter-declaration* is part of a *template-parameter-list* and the parameter pack is a template parameter pack; see 14.1. — *end note*] A function parameter pack, if present, shall occur at the end of the *parameter-declaration-list*. The type `T` of the *declarator-id* of the function parameter pack shall contain a template parameter pack; each template parameter pack in `T` is expanded by the function parameter pack. [*Example:*

```
template<typename... T> void f(T (* ...t)(int, int);

int add(int, int);
float subtract(int, int);

void g() {
  f(add, subtract);
}
```

— *end example*]

14 There is a syntactic ambiguity when an ellipsis occurs at the end of a *parameter-declaration-clause* without a preceding comma. In this case, the ellipsis is parsed as part of the *abstract-declarator* if the type of the

parameter names a template parameter pack that has not been expanded; otherwise, it is parsed as part of the *parameter-declaration-clause*.[91]

8.3.6 Default arguments [dcl.fct.default]

1 If an expression is specified in a parameter declaration this expression is used as a default argument. Default arguments will be used in calls where trailing arguments are missing.

2 [*Example:* the declaration

```
void point(int = 3, int = 4);
```

declares a function that can be called with zero, one, or two arguments of type `int`. It can be called in any of these ways:

```
point(1,2);  point(1);  point();
```

The last two calls are equivalent to `point(1,4)` and `point(3,4)`, respectively. — *end example*]

3 A default argument expression shall be specified only in the *parameter-declaration-clause* of a function declaration or in a *template-parameter* (14.1). It shall not be specified for a parameter pack. If it is specified in a *parameter-declaration-clause*, it shall not occur within a *declarator* or *abstract-declarator* of a *parameter-declaration*.[92]

4 For non-template functions, default arguments can be added in later declarations of a function in the same scope. Declarations in different scopes have completely distinct sets of default arguments. That is, declarations in inner scopes do not acquire default arguments from declarations in outer scopes, and vice versa. In a given function declaration, all parameters subsequent to a parameter with a default argument shall have default arguments supplied in this or previous declarations. A default argument shall not be redefined by a later declaration (not even to the same value). [*Example:*

```
void g(int = 0, ...);           // OK, ellipsis is not a parameter so it can follow
                                // a parameter with a default argument
void f(int, int);
void f(int, int = 7);
void h() {
  f(3);                         // OK, calls f(3, 7)
  void f(int = 1, int);         // error: does not use default
                                // from surrounding scope
}
void m() {
  void f(int, int);             // has no defaults
  f(4);                         // error: wrong number of arguments
  void f(int, int = 5);         // OK
  f(4);                         // OK, calls f(4, 5);
  void f(int, int = 5);         // error: cannot redefine, even to
                                // same value
}
void n() {
  f(6);                         // OK, calls f(6, 7)
}
```

91) One can explicitly disambiguate the parse either by introducing a comma (so the ellipsis will be parsed as part of the *parameter-declaration-clause*) or by introducing a name for the parameter (so the ellipsis will be parsed as part of the *declarator-id*).

92) This means that default arguments cannot appear, for example, in declarations of pointers to functions, references to functions, or `typedef` declarations.

— *end example*] For a given inline function defined in different translation units, the accumulated sets of default arguments at the end of the translation units shall be the same; see 3.2. If a friend declaration specifies a default argument expression, that declaration shall be a definition and shall be the only declaration of the function or function template in the translation unit.

5 A default argument expression is implicitly converted (Clause 4) to the parameter type. The default argument expression has the same semantic constraints as the initializer expression in a declaration of a variable of the parameter type, using the copy-initialization semantics (8.5). The names in the expression are bound, and the semantic constraints are checked, at the point where the default argument expression appears. Name lookup and checking of semantic constraints for default arguments in function templates and in member functions of class templates are performed as described in 14.7.1. [*Example:* in the following code, `g` will be called with the value `f(2)`:

```
int a = 1;
int f(int);
int g(int x = f(a));            // default argument: f(::a)

void h() {
  a = 2;
  {
  int a = 3;
  g();                          // g(f(::a))
  }
}
```

— *end example*] [*Note:* in member function declarations, names in default argument expressions are looked up as described in 3.4.1. Access checking applies to names in default argument expressions as described in Clause 11. — *end note*]

6 Except for member functions of class templates, the default arguments in a member function definition that appears outside of the class definition are added to the set of default arguments provided by the member function declaration in the class definition. Default arguments for a member function of a class template shall be specified on the initial declaration of the member function within the class template. [*Example:*

```
class C {
  void f(int i = 3);
  void g(int i, int j = 99);
};

void C::f(int i = 3) {          // error: default argument already
}                               // specified in class scope
void C::g(int i = 88, int j) {  // in this translation unit,
}                               // C::g can be called with no argument
```

— *end example*]

7 Local variables shall not be used in default argument expressions. [*Example:*

```
void f() {
  int i;
  extern void g(int x = i);     //error
  // ...
}
```

— *end example*]

8 The keyword `this` shall not be used in a default argument of a member function. [*Example:*

```
class A {
  void f(A* p = this) { }       // error
};
```

— *end example*]

9 Default arguments are evaluated each time the function is called. The order of evaluation of function arguments is unspecified. Consequently, parameters of a function shall not be used in default argument expressions, even if they are not evaluated. Parameters of a function declared before a default argument expression are in scope and can hide namespace and class member names. [*Example:*

```
int a;
int f(int a, int b = a);            // error: parameter a
                                    // used as default argument
typedef int I;
int g(float I, int b = I(2));       // error: parameter I found
int h(int a, int b = sizeof(a));    // error, parameter a used
                                    // in default argument
```

— *end example*] Similarly, a non-static member shall not be used in a default argument expression, even if it is not evaluated, unless it appears as the id-expression of a class member access expression (5.2.5) or unless it is used to form a pointer to member (5.3.1). [*Example:* the declaration of `X::mem1()` in the following example is ill-formed because no object is supplied for the non-static member `X::a` used as an initializer.

```
int b;
class X {
  int a;
  int mem1(int i = a);          // error: non-static member a
                                // used as default argument
  int mem2(int i = b);          // OK; use X::b
  static int b;
};
```

The declaration of `X::mem2()` is meaningful, however, since no object is needed to access the static member `X::b`. Classes, objects, and members are described in Clause 9. — *end example*] A default argument is not part of the type of a function. [*Example:*

```
int f(int = 0);

void h() {
  int j = f(1);
  int k = f();                  // OK, means f(0)
}

int (*p1)(int) = &f;
int (*p2)() = &f;               // error: type mismatch
```

— *end example*] When a declaration of a function is introduced by way of a *using-declaration* (7.3.3), any default argument information associated with the declaration is made known as well. If the function is redeclared thereafter in the namespace with additional default arguments, the additional arguments are also known at any point following the redeclaration where the *using-declaration* is in scope.

10 A virtual function call (10.3) uses the default arguments in the declaration of the virtual function determined by the static type of the pointer or reference denoting the object. An overriding function in a derived class does not acquire default arguments from the function it overrides. [*Example:*

```
struct A {
  virtual void f(int a = 7);
};
struct B : public A {
  void f(int a);
};
void m() {
  B* pb = new B;
  A* pa = pb;
  pa->f();          // OK, calls pa->B::f(7)
  pb->f();          // error: wrong number of arguments for B::f()
}
```

— *end example*]

8.4 Function definitions [dcl.fct.def]

1 Function definitions have the form

function-definition:
decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator function-body*
decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= default ;`
decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= delete ;`
function-body:
ctor-initializer$_{opt}$ *compound-statement*
function-try-block

Any informal reference to the body of a function should be interpreted as a reference to the non-terminal *function-body*.

2 The *declarator* in a *function-definition* shall have the form

`D1 (` *parameter-declaration-clause* `)` *cv-qualifier-seq*$_{opt}$ *ref-qualifier*$_{opt}$ *exception-specification*$_{opt}$

as described in 8.3.5. A function shall be defined only in namespace or class scope.

3 [*Example:* a simple example of a complete function definition is

```
int max(int a, int b, int c) {
  int m = (a > b) ? a : b;
  return (m > c) ? m : c;
}
```

Here `int` is the *decl-specifier-seq*; `max(int a, int b, int c)` is the *declarator*; `{ /* ... */ }` is the *function-body*. — *end example*]

4 A *ctor-initializer* is used only in a constructor; see 12.1 and 12.6.

5 A *cv-qualifier-seq* or a *ref-qualifier* (or both) can be part of a non-static member function declaration, non-static member function definition, or pointer to member function only (8.3.5); see 9.3.2.

6 [*Note:* unused parameters need not be named. For example,

```
void print(int a, int) {
  std::printf("a = %d\n",a);
}
```

— *end note*]

7 In the *function-body*, a *function-local predefined variable* denotes a local object of static storage duration that is implicitly defined (see 3.3.2).

8 The function-local predefined variable __func__ is defined as if a definition of the form

```
static const char __func__[] = "function-name";
```

had been provided, where *function-name* is an implementation-defined string. It is unspecified whether such a variable has an address distinct from that of any other object in the program.[93]

[*Example:*

```
struct S {
  S() : s(__func__) { }             // OK
  const char *s;
};
void f(const char * s = __func__);  // error: __func__ is undeclared
```

— *end example*]

9 A function definition of the form:

decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= default ;`

is called an *explicitly-defaulted* definition. Only special member functions may be explicitly defaulted, and the implementation shall define them as if they had implicit definitions (12.1, 12.4, 12.8). A special member function that would be implicitly defined as deleted shall not be explicitly defaulted. A special member function is *user-provided* if it is user-declared and not explicitly defaulted on its first declaration. A user-provided explicitly-defaulted function is defined at the point where it is explicitly defaulted. [*Note:* while an implicitly-declared special member function is inline (Clause 12), an explicitly-defaulted definition may be non-inline. Non-inline definitions are user-provided, and hence non-trivial (12.1, 12.4, 12.8). This rule enables efficient execution and concise definition while enabling a stable binary interface to an evolving code base. — *end note*] [*Example:*

```
struct trivial {
  trivial() = default;
  trivial(const trivial&) = default;
  trivial& operator =(const trivial&) = default;
  ~trivial() = default;
};

struct nontrivial1 {
  nontrivial1();
};
nontrivial1::nontrivial1() = default;           // not inline

struct nontrivial2 {
  nontrivial2();
};
inline nontrivial2::nontrivial2() = default;    // not first declaration

struct nontrivial3 {
  virtual ~nontrivial3() = 0;                   // virtual
};
inline nontrivial3::~nontrivial3() = default;   // not first declaration
```

93) Implementations are permitted to provide additional predefined variables with names that are reserved to the implementation (17.6.4.3.3). If a predefined variable is not used (3.2), its string value need not be present in the program image.

— *end example*]

10 A function definition of the form:

decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= delete ;`

is called a *deleted* definition. A function with a deleted definition is also called a *deleted function*. A deleted definition of a function shall be the first declaration of the function. [*Example:*

```
struct sometype {
  sometype();
};
sometype::sometype() = delete;   // ill-formed; not first declaration
```

— *end example*] A deleted function is implicitly inline. [*Note:* the one-definition rule (3.2) applies to deleted definitions. — *end note*] A program that refers to a deleted function implicitly or explicitly, other than to declare it, is ill-formed. [*Note:* this includes calling the function implicitly or explicitly and forming a pointer or pointer-to-member to the function. It applies even for references in expressions that are not potentially-evaluated. If a function is overloaded, it is referenced only if the function is selected by overload resolution. — *end note*] [*Example:* One can enforce non-default initialization and non-integral initialization with

```
struct sometype {
  sometype() = delete;           // redundant, but legal
  sometype(std::intmax_t) = delete;
  sometype(double);
};
```

— *end example*] [*Example:* One can prevent use of a class in certain `new` expressions by using deleted definitions of a user-declared `operator new` for that class.

```
struct sometype {
  void *operator new(std::size_t) = delete;
  void *operator new[](std::size_t) = delete;
};
sometype *p = new sometype;      // error, deleted class operator new
sometype *p = new sometype[3];   // error, deleted class operator new[]
```

— *end example*]

8.5 Initializers [dcl.init]

1 A declarator can specify an initial value for the identifier being declared. The identifier designates an object or reference being initialized. The process of initialization described in the remainder of 8.5 applies also to initializations specified by other syntactic contexts, such as the initialization of function parameters with argument expressions (5.2.2) or the initialization of return values (6.6.3).

initializer:
    brace-or-equal-initializer
    `(` *expression-list* `)`

brace-or-equal-initializer:
    `=` *initializer-clause*
    braced-init-list

initializer-clause:
    assignment-expression
    braced-init-list

initializer-list:
initializer-clause ...$_{opt}$
initializer-list , *initializer-clause* ...$_{opt}$

braced-init-list:
{ *initializer-list* ,$_{opt}$ }
{ }

2 Automatic, register, thread_local, static, and namespace-scoped external variables can be initialized by arbitrary expressions involving literals and previously declared variables and functions. [*Example:*

```
int f(int);
int a = 2;
int b = f(a);
int c(b);
```

— *end example*]

3 [*Note:* default argument expressions are more restricted; see 8.3.6.

4 The order of initialization of static objects is described in 3.6 and 6.7. — *end note*]

5 To *zero-initialize* an object or reference of type T means:

— if T is a scalar type (3.9), the object is set to the value 0 (zero), taken as an integral constant expression, converted to T;[94]

— if T is a (possibly cv-qualified) non-union class type, each non-static data member and each base-class subobject is zero-initialized;

— if T is a (possibly cv-qualified) union type, the object's first non-static named data member is zero-initialized;

— if T is an array type, each element is zero-initialized;

— if T is a reference type, no initialization is performed.

6 To *default-initialize* an object of type T means:

— if T is a (possibly cv-qualified) class type (Clause 9), the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);

— if T is an array type, each element is default-initialized;

— otherwise, no initialization is performed.

If a program calls for the default initialization of an object of a const-qualified type T, T shall be a class type with a user-provided default constructor.

7 To *value-initialize* an object of type T means:

— if T is a (possibly cv-qualified) class type (Clause 9) with a user-provided constructor (12.1), then the default constructor for T is called (and the initialization is ill-formed if T has no accessible default constructor);

— if T is a (possibly cv-qualified) non-union class type without a user-provided constructor, then the object is zero-initialized and, if T's implicitly-declared default constructor is non-trivial, that constructor is called.

— if T is an array type, then each element is value-initialized;

94) As specified in 4.10, converting an integral constant expression whose value is 0 to a pointer type results in a null pointer value.

— otherwise, the object is zero-initialized.

8 A program that calls for default-initialization or value-initialization of an entity of reference type is ill-formed.

9 [*Note:* Every object of static storage duration is zero-initialized at program startup before any other initialization takes place. in some cases, additional initialization is done later. — *end note*]

10 An object whose initializer is an empty set of parentheses, i.e., `()`, shall be value-initialized.

[*Note:* since `()` is not permitted by the syntax for *initializer*,

```
X a();
```

is not the declaration of an object of class `X`, but the declaration of a function taking no argument and returning an `X`. The form `()` is permitted in certain other initialization contexts (5.3.4, 5.2.3, 12.6.2). — *end note*]

11 If no initializer is specified for an object, the object is default-initialized; if no initialization is performed, a non-static object has indeterminate value. [*Note:* objects with static storage duration are zero-initialized, see 3.6.2 — *end note*].

12 An initializer for a static member is in the scope of the member's class. [*Example:*

```
int a;

struct X {
  static int a;
  static int b;
};

int X::a = 1;
int X::b = a;       // X::b = X::a
```

— *end example*]

13 The form of initialization (using parentheses or `=`) is generally insignificant, but does matter when the initializer or the entity being initialized has a class type; see below. A parenthesized initializer can be a list of expressions only when the entity being initialized has a class type.

14 The initialization that occurs in the form

```
T x = a;
```

as well as in argument passing, function return, throwing an exception (15.1), handling an exception (15.3), and aggregate member initialization (8.5.1) is called *copy-initialization*.

15 The initialization that occurs in in the forms

```
T x(a);
T x{a};
```

as well as in `new` expressions (5.3.4), `static_cast` expressions (5.2.9), functional notation type conversions (5.2.3), and base and member initializers (12.6.2) is called *direct-initialization*.

16 The semantics of initializers are as follows. The *destination type* is the type of the object or reference being initialized and the *source type* is the type of the initializer expression. The source type is not defined when the initializer is a *braced-init-list* or when it is a parenthesized list of expressions.

— If the destination type is a reference type, see 8.5.3.

— If the destination type is an array of characters, an array of `char16_t`, an array of `char32_t`, or an array of `wchar_t`, and the initializer is a string literal, see 8.5.2.

— If the initializer is a *braced-init-list*, the object is list-initialized (8.5.4).

— If the initializer is `()`, the object is value-initialized.

— Otherwise, if the destination type is an array, the program is ill-formed.

— If the destination type is a (possibly cv-qualified) class type:

  — If the initialization is direct-initialization, or if it is copy-initialization where the cv-unqualified version of the source type is the same class as, or a derived class of, the class of the destination, constructors are considered. The applicable constructors are enumerated (13.3.1.3), and the best one is chosen through overload resolution (13.3). The constructor so selected is called to initialize the object, with the initializer expression(s) as its argument(s). If no constructor applies, or the overload resolution is ambiguous, the initialization is ill-formed.

  — Otherwise (i.e., for the remaining copy-initialization cases), user-defined conversion sequences that can convert from the source type to the destination type or (when a conversion function is used) to a derived class thereof are enumerated as described in 13.3.1.4, and the best one is chosen through overload resolution (13.3). If the conversion cannot be done or is ambiguous, the initialization is ill-formed. The function selected is called with the initializer expression as its argument; if the function is a constructor, the call initializes a temporary of the cv-unqualified version of the destination type. The temporary is an rvalue. The result of the call (which is the temporary for the constructor case) is then used to direct-initialize, according to the rules above, the object that is the destination of the copy-initialization. In certain cases, an implementation is permitted to eliminate the copying inherent in this direct-initialization by constructing the intermediate result directly into the object being initialized; see 12.2, 12.8.

— Otherwise, if the source type is a (possibly cv-qualified) class type, conversion functions are considered. The applicable conversion functions are enumerated (13.3.1.5), and the best one is chosen through overload resolution (13.3). The user-defined conversion so selected is called to convert the initializer expression into the object being initialized. If the conversion cannot be done or is ambiguous, the initialization is ill-formed.

— Otherwise, the initial value of the object being initialized is the (possibly converted) value of the initializer expression. Standard conversions (Clause 4) will be used, if necessary, to convert the initializer expression to the cv-unqualified version of the destination type; no user-defined conversions are considered. If the conversion cannot be done, the initialization is ill-formed. [*Note:* an expression of type "*cv1* `T`" can initialize an object of type "*cv2* `T`" independently of the cv-qualifiers *cv1* and *cv2*.

```
int a;
const int b = a;
int c = b;
```

— *end note*]

17 An *initializer-clause* followed by an ellipsis is a pack expansion (14.5.3).

8.5.1 Aggregates [dcl.init.aggr]

1 An *aggregate* is an array or a class (Clause 9) with no user-provided constructors (12.1), no private or protected non-static data members (Clause 11), no base classes (Clause 10), and no virtual functions (10.3).

2 When an aggregate is initialized by an initializer list, as specified in 8.5.4, the elements of the initializer list are taken as initializers for the members of the aggregate, in increasing subscript or member order. Each member is copy-initialized from the corresponding *initializer-clause*. If the *initializer-clause* is an expression and a narrowing conversion (8.5.4) is required to convert the expression, the program is ill-formed. [*Note:* If an *initializer-clause* is itself an initializer list, the member is list-initialized, which will result in a recursive application of the rules in this section if the member is an aggregate. — *end note*] [*Example:*

```
struct A {
  int x;
  struct B {
    int i;
    int j;
  } b;
} a = { 1, { 2, 3 } };
```

initializes `a.x` with 1, `a.b.i` with 2, `a.b.j` with 3. — *end example*]

3 An aggregate that is a class can also be initialized with a single expression not enclosed in braces, as described in 8.5.

4 An array of unknown size initialized with a brace-enclosed *initializer-list* containing `n` *initializer-clause*s, where `n` shall be greater than zero, is defined as having `n` elements (8.3.4). [*Example:*

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array that has three elements since no size was specified and there are three initializers. — *end example*] An empty initializer list `{}` shall not be used as the *initializer-clause* for an array of unknown bound.[95]

5 Static data members and anonymous bit fields are not considered members of the class for purposes of aggregate initialization. [*Example:*

```
struct A {
  int i;
  static int s;
  int j;
  int :17;
  int k;
} a = { 1, 2, 3 };
```

Here, the second initializer 2 initializes `a.j` and not the static data member `A::s`, and the third initializer 3 initializes `a.k` and not the anonymous bit field before it. — *end example*]

6 An *initializer-list* is ill-formed if the number of *initializer-clause*s exceeds the number of members or elements to initialize. [*Example:*

```
char cv[4] = { 'a', 's', 'd', 'f', 0 };      // error
```

is ill-formed. — *end example*]

7 If there are fewer *initializer-clause*s in the list than there are members in the aggregate, then each member not explicitly initialized shall be value-initialized (8.5). [*Example:*

```
struct S { int a; char* b; int c; };
S ss = { 1, "asdf" };
```

95) The syntax provides for empty *initializer-list*s, but nonetheless C++ does not have zero length arrays.

initializes `ss.a` with 1, `ss.b` with `"asdf"`, and `ss.c` with the value of an expression of the form `int()`, that is, `0`. — *end example*]

8 If an aggregate class `C` contains a subaggregate member `m` that has no members for purposes of aggregate initialization, the *initializer-clause* for `m` shall not be omitted from an *initializer-list* for an object of type `C` unless the *initializer-clause*s for all members of `C` following `m` are also omitted. [*Example:*

```
struct S { } s;
struct A {
  S s1;
  int i1;
  S s2;
  int i2;
  S s3;
  int i3;
} a = {
  { },       // Required initialization
  0,
  s,         // Required initialization
  0
};           // Initialization not required for A::s3 because A::i3 is also not initialized
```

— *end example*]

9 If an incomplete or empty *initializer-list* leaves a member of reference type uninitialized, the program is ill-formed.

10 When initializing a multi-dimensional array, the *initializer-clause*s initialize the elements with the last (rightmost) index of the array varying the fastest (8.3.4). [*Example:*

```
int x[2][2] = { 3, 1, 4, 2 };
```

initializes `x[0][0]` to `3`, `x[0][1]` to `1`, `x[1][0]` to `4`, and `x[1][1]` to `2`. On the other hand,

```
float y[4][3] = {
  { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest zero. — *end example*]

11 In a declaration of the form

```
T x = { a };
```

braces can be elided in an *initializer-list* as follows.[96] If the *initializer-list* begins with a left brace, then the succeeding comma-separated list of *initializer-clause*s initializes the members of a subaggregate; it is erroneous for there to be more *initializer-clause*s than members. If, however, the *initializer-list* for a subaggregate does not begin with a left brace, then only enough *initializer-clause*s from the list are taken to initialize the members of the subaggregate; any remaining *initializer-clause*s are left to initialize the next member of the aggregate of which the current subaggregate is a member. [*Example:*

```
float y[4][3] = {
  { 1, 3, 5 },
  { 2, 4, 6 },
  { 3, 5, 7 },
};
```

96) Braces cannot be elided in other uses of list-initialization.

is a completely-braced initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise the next two lines initialize `y[1]` and `y[2]`. The initializer ends early and therefore `y[3]`s elements are initialized as if explicitly initialized with an expression of the form `float()`, that is, are initialized with `0.0`. In the following example, braces in the *initializer-list* are elided; however the *initializer-list* has the same effect as the completely-braced *initializer-list* of the above example,

```
float y[4][3] = {
  1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace, but the one for `y[0]` does not, therefore three elements from the list are used. Likewise the next three are taken successively for `y[1]` and `y[2]`. — *end example*]

12 All implicit type conversions (Clause 4) are considered when initializing the aggregate member with an *assignment-expression*. If the *assignment-expression* can initialize a member, the member is initialized. Otherwise, if the member is itself a subaggregate, brace elision is assumed and the *assignment-expression* is considered for the initialization of the first member of the subaggregate. [*Note:* As specified above, brace elision cannot apply to subaggregates with no members for purposes of aggregate initialization; an *initializer-clause* for the entire subobject is required. — *end note*]

[*Example:*

```
struct A {
  int i;
  operator int();
};
struct B {
  A a1, a2;
  int z;
};
A a;
B b = { 4, a, a };
```

Braces are elided around the *initializer-clause* for `b.a1.i`. `b.a1.i` is initialized with 4, `b.a2` is initialized with `a`, `b.z` is initialized with whatever `a.operator int()` returns. — *end example*]

13 [*Note:* An aggregate array or an aggregate class may contain members of a class type with a user-provided constructor (12.1). Initialization of these aggregate objects is described in 12.6.1. — *end note*]

14 [*Note:* Whether the initialization of aggregates with static storage duration is static or dynamic is specified in 3.6.2 and 6.7. — *end note*]

15 When a union is initialized with a brace-enclosed initializer, the braces shall only contain an *initializer-clause* for the first member of the union. [*Example:*

```
union u { int a; char* b; };
u a = { 1 };
u b = a;
u c = 1;                        // error
u d = { 0, "asdf" };            // error
u e = { "asdf" };               // error
```

— *end example*]

16 [*Note:* As described above, the braces around the *initializer-clause* for a union member can be omitted if the union is a member of another aggregate. — *end note*]

17 The full-expressions in an *initializer-clause* are evaluated in the order in which they appear.

8.5.2 Character arrays [dcl.init.string]

1 A `char` array (whether plain `char`, `signed char`, or `unsigned char`), `char16_t` array, `char32_t` array, or `wchar_t` array can be initialized by a *string-literal* (optionally enclosed in braces) with no prefix, with a `u` prefix, with a `U` prefix, or with an `L` prefix, respectively; successive characters of the *string-literal* initialize the members of the array. [*Example:*

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a *string-literal*. Note that because `'\n'` is a single character and because a trailing `'\0'` is appended, `sizeof(msg)` is `25`. — *end example*]

2 There shall not be more initializers than there are array elements. [*Example:*

```
char cv[4] = "asdf";            // error
```

is ill-formed since there is no space for the implied trailing `'\0'`. — *end example*]

8.5.3 References [dcl.init.ref]

1 A variable declared to be a `T&` or `T&&`, that is, "reference to type `T`" (8.3.2), shall be initialized by an object, or function, of type `T` or by an object that can be converted into a `T`. [*Example:*

```
int g(int);
void f() {
  int i;
  int& r = i;                   // r refers to i
  r = 1;                        // the value of i becomes 1
  int* p = &r;                  // p points to i
  int& rr = r;                  // rr refers to what r refers to, that is, to i
  int (&rg)(int) = g;           // rg refers to the function g
  rg(i);                        // calls function g
  int a[3];
  int (&ra)[3] = a;             // ra refers to the array a
  ra[1] = i;                    // modifies a[1]
}
```

— *end example*]

2 A reference cannot be changed to refer to another object after initialization. Note that initialization of a reference is treated very differently from assignment to it. Argument passing (5.2.2) and function value return (6.6.3) are initializations.

3 The initializer can be omitted for a reference only in a parameter declaration (8.3.5), in the declaration of a function return type, in the declaration of a class member within its class definition (9.2), and where the `extern` specifier is explicitly used. [*Example:*

```
int& r1;                        // error: initializer missing
extern int& r2;                 // OK
```

— *end example*]

4 Given types "*cv1* `T1`" and "*cv2* `T2`," "*cv1* `T1`" is *reference-related* to "*cv2* `T2`" if `T1` is the same type as `T2`, or `T1` is a base class of `T2`. "*cv1* `T1`" is *reference-compatible* with "*cv2* `T2`" if `T1` is reference-related to `T2` and *cv1* is the same cv-qualification as, or greater cv-qualification than, *cv2*. For purposes of overload resolution,

cases for which *cv1* is greater cv-qualification than *cv2* are identified as *reference-compatible with added qualification* (see 13.3.3.2). In all cases where the reference-related or reference-compatible relationship of two types is used to establish the validity of a reference binding, and `T1` is a base class of `T2`, a program that necessitates such a binding is ill-formed if `T1` is an inaccessible (Clause 11) or ambiguous (10.2) base class of `T2`.

5 A reference to type "*cv1* `T1`" is initialized by an expression of type "*cv2* `T2`" as follows:

— If the initializer expression

— is an lvalue (but is not a bit-field), and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," or

— has a class type (i.e., `T2` is a class type) and can be implicitly converted to an lvalue of type "*cv3* `T3`," where "*cv1* `T1`" is reference-compatible with "*cv3* `T3`"[97] (this conversion is selected by enumerating the applicable conversion functions (13.3.1.6) and choosing the best one through overload resolution (13.3)),

then the reference is bound directly to the initializer expression lvalue in the first case, and the reference is bound to the lvalue result of the conversion in the second case. In these cases the reference is said to *bind directly* to the initializer expression. [*Note:* the usual lvalue-to-rvalue (4.1), array-to-pointer (4.2), and function-to-pointer (4.3) standard conversions are not needed, and therefore are suppressed, when such direct bindings to lvalues are done. — *end note*]

[*Example:*

```
double d = 2.0;
double& rd = d;                 // rd refers to d
const double& rcd = d;          // rcd refers to d

struct A { };
struct B : A { } b;
A& ra = b;                      // ra refers to A subobject in b
const A& rca = b;               // rca refers to A subobject in b
```

— *end example*]

— Otherwise, the reference shall be an lvalue reference to a non-volatile const type (i.e., *cv1* shall be `const`), or shall be an rvalue reference. [*Example:*

```
double& rd2 = 2.0;              // error: not an lvalue and reference not const
int  i = 2;
double& rd3 = i;                // error: type mismatch and reference not const
double&& rd4 = i;               // OK: reference bound to temporary double
```

— *end example*]

— If the initializer expression is an rvalue, with `T2` a class type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," the reference is bound to the object represented by the rvalue (see 3.10) or to a sub-object within that object.

[*Example:*

```
struct A { };
struct B : A { } b;
extern B f();
const A& rca = f();             // Bound to the A subobject of the B rvalue.
A&& rcb = f();                  // Same as above
```

97) This requires a conversion function (12.3.2) returning a reference type.

— *end example*]

— If the initializer expression is an rvalue, with `T2` an array type, and "*cv1* `T1`" is reference-compatible with "*cv2* `T2`," the reference is bound to the object represented by the rvalue (see 3.10).

— Otherwise, a temporary of type "*cv1* `T1`" is created and initialized from the initializer expression using the rules for a non-reference copy initialization (8.5). The reference is then bound to the temporary. If `T1` is reference-related to `T2`, *cv1* must be the same cv-qualification as, or greater cv-qualification than, *cv2*; otherwise, the program is ill-formed. [*Example:*

```
const double& rcd2 = 2;          // rcd2 refers to temporary with value 2.0
double&& rcd3 = 2;               // rcd3 refers to temporary with value 2.0
const volatile int cvi = 1;
const int& r = cvi;              // error: type qualifiers dropped
```

— *end example*]

6 [*Note:* 12.2 describes the lifetime of temporaries bound to references. — *end note*]

8.5.4 List-initialization [dcl.init.list]

1 *List-initialization* is initialization of an object or reference from a *braced-init-list*. Such an initializer is called an *initializer list*, and the comma-separated *initializer-clause*s of the list are called the *elements* of the initializer list. An initializer list may be empty. List-initialization can occur in direct-initialization or copy-initialization contexts; list-initialization in a direct-initialization context is called *direct-list-initialization* and list-initialization in a copy-initialization context is called *copy-list-initialization*. [*Note:* List-initialization can be used

— as the initializer in a variable definition (8.5)

— as the initializer in a new expression (5.3.4)

— in a return statement (6.6.3)

— as a function argument (5.2.2)

— as a subscript (5.2.1)

— as an argument to a constructor invocation (8.5, 5.2.3)

— as an initializer for a non-static data member (9.2)

— as a base-or-member initializer (12.6.2)

— on the right-hand side of an assignment (5.17)

[*Example:*

```
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once", "upon", "a", "time"};  // 4 string elements
f( {"Nicholas","Annemarie"} );  // pass list of two elements
return { "Norah" };             // return list of one element
int* e {};                      // initialization to zero / null pointer
x = double{1};                  // explicitly construct a double
std::map<std::string,int> anim = { {"bear",4}, {"cassowary",2}, {"tiger",7} };
```

— *end example*] — *end note*]

2 A constructor is an *initializer-list constructor* if its first parameter is of type `std::initializer_list<E>` or reference to possibly cv-qualified `std::initializer_list<E>` for some type `E`, and either there are no other parameters or else all other parameters have default arguments (8.3.6). [*Note:* Initializer-list constructors are favored over other constructors in list-initialization (13.3.1.7). — *end note*] The template `std::initializer_list` is not predefined; if the header `<initializer_list>` is not included prior to a use of `std::initializer_list` — even an implicit use in which the type is not named (7.1.6.4) — the program is ill-formed.

3 List-initialization of an object or reference of type `T` is defined as follows:

— If `T` is an aggregate, aggregate initialization is performed (8.5.1).

[*Example:*

```
double ad[] = { 1, 2.0 };           // OK
int ai[] = { 1, 2.0 };              // error: narrowing
```

— *end example*]

— Otherwise, if `T` is a specialization of `std::initializer_list<E>`, an `initializer_list` object is constructed as described below and used to initialize the object according to the rules for initialization of an object from a class of the same type (8.5).

— Otherwise, if `T` is a class type, constructors are considered. If `T` has an initializer-list constructor, the argument list consists of the initializer list as a single argument; otherwise, the argument list consists of the elements of the initializer list. The applicable constructors are enumerated (13.3.1.7) and the best one is chosen through overload resolution (13.3). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

[*Example:*

```
struct S {
  S(std::initializer_list<double>); // #1
  S(std::initializer_list<int>);    // #2
  // ...
};
S s1 = { 1.0, 2.0, 3.0 };           // invoke #1
S s2 = { 1, 2, 3 };                 // invoke #2
```

— *end example*]

[*Example:*

```
struct Map {
  Map(std::initializer_list<std::pair<std::string,int>>);
};
Map ship = {{"Sophie",14}, {"Surprise",28}};
```

— *end example*]

[*Example:*

```
struct S {
  // no initializer-list constructors
  S(int, double, double);           // #1
  S();                              // #2
  // ...
};
```

```
S s1 = { 1, 2, 3.0 };              // OK: invoke #1
S s2 { 1.0, 2, 3 };                // error: narrowing
S s3 { };                          // OK: invoke #2

struct S2 {
  int m1;
  double m2,m3;
};
S2 s21 = { 1, 2, 3.0 };            // OK
S2 s22 { 1.0, 2, 3 };              // error: narrowing
S2 s23 {};                         // OK: default to 0,0,0
```

— *end example*]

— Otherwise, if `T` is a reference type, an rvalue temporary of the type referenced by `T` is list-initialized, and the reference is bound to that temporary. [*Note:* As usual, the binding will fail and the program is ill-formed if the reference type is an lvalue reference to a non-const type. — *end note*]

[*Example:*

```
struct S {
  S(std::initializer_list<double>); // #1
  S(const std::string&);            // #2
  // ...
};
const S& r1 = { 1, 2, 3.0 };       // OK: invoke #1
const S& r2 { "Spinach" };         // OK: invoke #2
S& r3 = { 1, 2, 3 };               // error: initializer is not an lvalue
```

— *end example*]

— Otherwise (i.e., if `T` is not an aggregate, class type, or reference), if the initializer list has a single element, the object is initialized from that element; if a narrowing conversion (see below) is required to convert the element to `T`, the program is ill-formed.

[*Example:*

```
int x1 {2};                        // OK
int x2 {2.0};                      // error: narrowing
```

— *end example*]

— Otherwise, if the initializer list has no elements, the object is value-initialized.

[*Example:*

```
int** pp {};                       // initialized to null pointer
```

— *end example*]

— Otherwise, the program is ill-formed.

[*Example:*

```
struct A { int i; int j; };
A a1 { 1, 2 };                     // aggregate initialization
A a2 { 1.2 };                      // error: narrowing
struct B {
  B(std::initializer_list<int>);
```

```
};
B b1 { 1, 2 };                  // creates initializer_list<int> and calls constructor
B b2 { 1, 2.0 };                // error: narrowing
struct C {
  C(int i, double j);
};
C c1 = { 1, 2.2 };              // calls constructor with arguments (1, 2.2)
C c2 = { 1.1, 2 };              // error: narrowing

int j { 1 };                    // initialize to 1
int k { };                      // initialize to 0
```

— *end example*]

4 When an initializer list is implicitly converted to a `std::initializer_list<E>`, the object passed is constructed as if the implementation allocated an array of N elements of type `E`, where N is the number of elements in the initializer list. Each element of that array is initialized with the corresponding element of the initializer list converted to `E`, and the `std::initializer_list<E>` object is constructed to refer to that array. If a narrowing conversion is required to convert the element to `E`, the program is ill-formed.[*Example:*

```
struct X {
  X(std::initializer_list<double> v);
};
X x{ 1,2,3 };
```

The initialization will be implemented in a way roughly equivalent to this:

```
double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

assuming that the implementation can construct an `initializer_list` object with a pair of pointers. — *end example*]

5 The lifetime of the array is the same as that of the `initializer_list` object. [*Example:*

```
typedef std::complex<double> cmplx;
std::vector<cmplx> v1 = { 1, 2, 3 };

void f() {
  std::vector<cmplx> v2{ 1, 2, 3 };
  std::initializer_list<int> i3 = { 1, 2, 3 };
}
```

For `v1` and `v2`, the `initializer_list` object and array created for `{ 1, 2, 3 }` have full-expression lifetime. For `i3`, the `initializer_list` object and array have automatic lifetime. — *end example*] [*Note:* The implementation is free to allocate the array in read-only memory if an explicit array with the same initializer could be so allocated. — *end note*]

6 A *narrowing conversion* is an implicit conversion

— from a floating-point type to an integer type, or

— from `long double` to `double` or `float`, or from `double` to `float`, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

— from an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type, or

— from an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted back to the original type.

[*Note:* As indicated above, such conversions are not allowed at the top level in list-initializations. [*Example:*

```
int x = 999;                // x is not a constant expression
const int y = 999;
const int z = 99;
char c1 = x;                // OK, though it might narrow (in this case, it does narrow)
char c2{x};                 // error, might narrow
char c3{y};                 // error: narrows
char c4{z};                 // OK, no narrowing needed
unsigned char uc1 = {5};    // OK: no narrowing needed
unsigned char uc2 = {-1}; // error: narrows
unsigned int ui1 = {-1};  // error: narrows
signed int si1 =
  { (unsigned int)-1 };     // error: narrows
int ii = {2.0};             // error: narrows
float f1 { x };             // error: narrows
float f2 { 7 };             // OK: 7 can be exactly represented as a float
int f(int);
int a[] =
  { 2, f(2), f(2.0) };      // OK: the double-to-int conversion is not at the top level
```

— *end example*]

9 Classes [class]

1 A class is a type. Its name becomes a *class-name* (9.1) within its scope.

class-name:
identifier
simple-template-id

*Class-specifier*s and *elaborated-type-specifier*s (7.1.6.3) are used to make *class-name*s. An object of a class consists of a (possibly empty) sequence of members and base class objects.

class-specifier:
class-head `{` *member-specification*$_{opt}$ `}`

class-head:
class-key identifier$_{opt}$ *attribute-specifier*$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier identifier attribute-specifier$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier$_{opt}$ *simple-template-id attribute-specifier*$_{opt}$ *base-clause*$_{opt}$

class-key:
`class`
`struct`
`union`

A *class-specifier* where the *class-head* omits the optional *identifier* defines an unnamed class.

2 A *class-name* is inserted into the scope in which it is declared immediately after the *class-name* is seen. The *class-name* is also inserted into the scope of the class itself; this is known as the *injected-class-name*. For purposes of access checking, the injected-class-name is treated as if it were a public member name. A *class-specifier* is commonly referred to as a class definition. A class is considered defined after the closing brace of its *class-specifier* has been seen even though its member functions are in general not yet defined. The optional *attribute-specifier* appertains to the class; the attributes in the *attribute-specifier* are thereafter considered attributes of the class whenever it is named.

3 Complete objects and member subobjects of class type shall have nonzero size.[98] [*Note:* class objects can be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see 12.8). Other plausible operators, such as equality comparison, can be defined by the user; see 13.5. — *end note*]

4 A *union* is a class defined with the *class-key* `union`; it holds only one data member at a time (9.5). [*Note:* aggregates of class type are described in 8.5.1. — *end note*]

5 An *effective class* `T` is a non-archetype class or a type archetype (14.10.2) that has the requirement `std::ClassType<T>`.

6 A *trivially copyable class* is a class that:

— has no non-trivial copy constructors (12.8),

— has no non-trivial copy assignment operators (13.5.3, 12.8), and

— has a trivial destructor (12.4).

98) Base class subobjects are not so constrained.

A *trivial class* is a class that has a trivial default constructor (12.1) and is trivially copyable.

[*Note:* in particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. — *end note*]

7 A *standard-layout class* is a class that:

— has no non-static data members of type non-standard-layout class (or array of such types) or reference,

— has no virtual functions (10.3) and no virtual base classes (10.1),

— has the same access control (Clause 11) for all non-static data members,

— has no non-standard-layout base classes,

— either has no non-static data members in the most-derived class and at most one base class with non-static data members, or has no base classes with non-static data members, and

— has no base classes of the same type as the first non-static data member.[99]

8 A *standard-layout struct* is a standard-layout class defined with the *class-key* `struct` or the *class-key* `class`. A *standard-layout union* is a standard-layout class defined with the *class-key* `union`.

9 [*Note:* standard-layout classes are useful for communicating with code written in other programming languages. Their layout is specified in 9.2. — *end note*]

10 A *POD struct* is a class that is both a trivial class and a standard-layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). Similarly, a *POD union* is a union that is both a trivial class and a standard layout class, and has no non-static data members of type non-POD struct, non-POD union (or array of such types). A *POD class* is a class that is either a POD struct or a POD union.

[*Example:*

```
struct N {          // neither trivial nor standard-layout
  int i;
  int j;
  virtual ~N();
};

struct T {          // trivial but not standard-layout
  int i;
private:
  int j;
};

struct SL {         // standard-layout but not trivial
  int i;
  int j;
  ~SL();
};

struct POD {        // both trivial and standard-layout
  int i;
  int j;
};
```

99) This ensures that two subobjects that have the same class type and that belong to the same most-derived object are not allocated at the same address (5.10).

— *end example*]

11 If a *class-head* contains a *nested-name-specifier*, the *class-specifier* shall refer to a class that was previously declared directly in the class or namespace to which the *nested-name-specifier* refers (i.e., neither inherited nor introduced by a *using-declaration*), and the *class-specifier* shall appear in a namespace enclosing the previous declaration.

9.1 Class names [class.name]

1 A class definition introduces a new type. [*Example:*

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```
a1 = a2;                    // error: Y assigned to X
a1 = a3;                    // error: int assigned to X
```

are type mismatches, and that

```
int f(X);
int f(Y);
```

declare an overloaded (Clause 13) function `f()` and not simply a single function `f()` twice. For the same reason,

```
struct S { int a; };
struct S { int a; };        // error, double definition
```

is ill-formed because it defines `S` twice. — *end example*]

2 A class declaration introduces the class name into the scope where it is declared and hides any class, object, function, or other declaration of that name in an enclosing scope (3.3). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared, then when both declarations are in scope, the class can be referred to only using an *elaborated-type-specifier* (3.4.4). [*Example:*

```
struct stat {
  // ...
};

stat gstat;                 // use plain stat to
                            // define variable

int stat(struct stat*);     // redeclare stat as function

void f() {
  struct stat* ps;          // struct prefix needed
                            // to name struct stat
  stat(ps);                 // call stat()
}
```

— *end example*] A *declaration* consisting solely of *class-key identifier;* is either a redeclaration of the name in the current scope or a forward declaration of the identifier as a class name. It introduces the class name into the current scope. [*Example:*

```
struct s { int a; };

void g() {
  struct s;                     // hide global struct s
                                // with a local declaration
  s* p;                         // refer to local struct s
  struct s { char* p; };        // define local struct s
  struct s;                     // redeclaration, has no effect
}
```

— *end example*] [*Note:* Such declarations allow definition of classes that refer to each other. [*Example:*

```
class Vector;

class Matrix {
  // ...
  friend Vector operator*(const Matrix&, const Vector&);
};

class Vector {
  // ...
  friend Vector operator*(const Matrix&, const Vector&);
};
```

Declaration of `friend`s is described in 11.4, operator functions in 13.5. — *end example*] — *end note*]

3 [*Note:* An *elaborated-type-specifier* (7.1.6.3) can also be used as a *type-specifier* as part of a declaration. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. — *end note*] [*Example:*

```
struct s { int a; };

void g(int s) {
  struct s* p = new struct s;   // global s
  p->a = s;                     // local s
}
```

— *end example*]

4 [*Note:* The declaration of a class name takes effect immediately after the *identifier* is seen in the class definition or *elaborated-type-specifier*. For example,

```
class A * A;
```

first specifies `A` to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form `class A` must be used to refer to the class. Such artistry with names can be confusing and is best avoided. — *end note*]

5 A *typedef-name* (7.1.3) that names a class type, or a cv-qualified version thereof, is also a *class-name*. If a *typedef-name* that names a cv-qualified class type is used where a *class-name* is required, the cv-qualifiers

are ignored. A *typedef-name* shall not be used as the *identifier* in a *class-head*.

9.2 Class members [class.mem]

member-specification:
member-declaration member-specification$_{opt}$
access-specifier : *member-specification*$_{opt}$

member-declaration:
member-requirement$_{opt}$ *decl-specifier-seq*$_{opt}$
attribute-specifier$_{opt}$ *member-declarator-list*$_{opt}$;
member-requirement$_{opt}$ *function-definition* ;$_{opt}$
::$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id* ;
using-declaration
static_assert-declaration
template-declaration

member-requirement:
requires-clause

member-declarator-list:
member-declarator
member-declarator-list , *member-declarator*

member-declarator:
declarator pure-specifier$_{opt}$
declarator brace-or-equal-initializer$_{opt}$
identifier$_{opt}$ *attribute-specifier*$_{opt}$: *constant-expression*

pure-specifier:
`= 0`

1 The *member-specification* in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (9.3), nested types, and enumerators. Data members and member functions are static or non-static; see 9.4. Nested types are classes (9.1, 9.7) and enumerations (7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (7.1.3). The enumerators of an enumeration (7.2) defined in the class are members of the class. Except when used to declare friends (11.4) or to introduce the name of a member of a base class into a derived class (7.3.3, 11.3), *member-declaration*s declare members of the class, and each such *member-declaration* shall declare at least one member name of the class. A member shall not be declared twice in the *member-specification*, except that a nested class or member class template can be declared and then later defined.

2 A non-archetype class is considered a completely-defined object type (3.9) (or complete type) at the closing } of the *class-specifier*. [*Note:* Each type archetype (14.10.2) is considered to be a unique generated class type and is considered to be defined when it is established. — *end note*] Within the class *member-specification*, the class is regarded as complete within function bodies, default arguments, *exception-specification*s, and *brace-or-equal-initializer*s for non-static data members (including such things in nested classes). Otherwise it is regarded as incomplete within its own class *member-specification*.

3 [*Note:* a single name can denote several function members provided their types are sufficiently different (Clause 13). — *end note*]

4 A member can be initialized using a constructor; see 12.1. [*Note:* see Clause 12 for a description of constructors and other special member functions. — *end note*]

5 A member can be initialized using a *brace-or-equal-initializer*. (For static data members, see 9.4.2; for non-static data members, see 12.6.2).

6 A member shall not be declared with the `extern` or `register` *storage-class-specifier*. Within a class definition, a member shall not be declared with the `thread_local` *storage-class-specifier* unless also declared `static`.

7 The *decl-specifier-seq* is omitted in constructor, destructor, and conversion function declarations only. The *member-declarator-list* can be omitted only after a *class-specifier* or an *enum-specifier* or in a `friend` declaration (11.4). A *pure-specifier* shall be used only in the declaration of a virtual function (10.3).

8 Non-`static` (9.4) data members shall not have incomplete types. In particular, a class `C` shall not contain a non-static member of class `C`, but it can contain a pointer or reference to an object of class `C`.

9 [*Note:* See 5.1 for restrictions on the use of non-static data members and non-static member functions. — *end note*]

10 [*Note:* the type of a non-static member function is an ordinary function type, and the type of a non-static data member is an ordinary object type. There are no special member function types or data member types. — *end note*]

11 [*Example:* A simple example of a class definition is

```
struct tnode {
  char tword[20];
  int count;
  tnode *left;
  tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to objects of the same type. Once this definition has been given, the declaration

```
tnode s, *sp;
```

declares `s` to be a `tnode` and `sp` to be a pointer to a `tnode`. With these declarations, `sp->count` refers to the `count` member of the object to which `sp` points; `s.left` refers to the `left` subtree pointer of the object `s`; and `s.right->tword[0]` refers to the initial character of the `tword` member of the `right` subtree of `s`. — *end example*]

12 Nonstatic data members of a (non-union) class with the same access control (Clause 11) are allocated so that later members have higher addresses within a class object. The order of allocation of non-static data members with different access control is unspecified (11). Implementation alignment requirements might cause two adjacent members not to be allocated immediately after each other; so might requirements for space for managing virtual functions (10.3) and virtual base classes (10.1).

13 If `T` is the name of a class, then each of the following shall have a name different from `T`:

- every static data member of class `T`;
- every member function of class `T` [*Note:* this restriction does not apply to constructors, which do not have names (12.1) — *end note*];
- every member of class `T` that is itself a type;
- every enumerator of every member of class `T` that is an enumerated type; and
- every member of every anonymous union that is a member of class `T`.

14 In addition, if class `T` has a user-declared constructor (12.1), every non-static data member of class `T` shall have a name different from `T`.

15 Two standard-layout struct (Clause 9) types are layout-compatible if they have the same number of non-static data members and corresponding non-static data members (in declaration order) have layout-compatible types (3.9).

16 Two standard-layout union (Clause 9) types are layout-compatible if they have the same number of non-static data members and corresponding non-static data members (in any order) have layout-compatible types (3.9).

17 If a standard-layout union contains two or more standard-layout structs that share a common initial sequence, and if the standard-layout union object currently contains one of these standard-layout structs, it is permitted to inspect the common initial part of any of them. Two standard-layout structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

18 A pointer to a standard-layout struct object, suitably converted using a `reinterpret_cast`, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa. [*Note:* There might therefore be unnamed padding within a standard-layout struct object, but not at its beginning, as necessary to achieve appropriate alignment. — *end note*]

19 A non-template *member-declaration* that has a *member-requirement* (14.10.1) is a *constrained member* and shall occur only in a class template (14.5.1) or nested class thereof. The *member-declaration* for a constrained member shall declare a member function. A constrained member is treated as a constrained template (14.10) whose template requirements include the requirements specified in its *member-requirement* clause and the requirements of each enclosing constrained template.

9.3 Member functions [class.mfct]

1 Functions declared in the definition of a class, excluding those declared with a `friend` specifier (11.4), are called member functions of that class. A member function may be declared `static` in which case it is a *static* member function of its class (9.4); otherwise it is a *non-static* member function of its class (9.3.1, 9.3.2).

2 A member function may be defined (8.4) in its class definition, in which case it is an *inline* member function (7.1.2), or it may be defined outside of its class definition if it has already been declared but not defined in its class definition. A member function definition that appears outside of the class definition shall appear in a namespace scope enclosing the class definition. Except for member function definitions that appear outside of a class definition, and except for explicit specializations of member functions of class templates and member function templates (14.7) appearing outside of the class definition, a member function shall not be redeclared.

3 An `inline` member function (whether static or non-static) may also be defined outside of its class definition provided either its declaration in the class definition or its definition outside of the class definition declares the function as `inline`. [*Note:* member functions of a class in namespace scope have external linkage. Member functions of a local class (9.8) have no linkage. See 3.5. — *end note*]

4 There shall be at most one definition of a non-inline member function in a program; no diagnostic is required. There may be more than one `inline` member function definition in a program. See 3.2 and 7.1.2.

5 If the definition of a member function is lexically outside its class definition, the member function name shall be qualified by its class name using the `::` operator. [*Note:* a name used in a member function definition (that is, in the *parameter-declaration-clause* including the default arguments (8.3.6) or in the member function body) is looked up as described in 3.4. — *end note*] [*Example:*

```
struct X {
  typedef int T;
  static T count;
```

```
  void f(T);
};
void X::f(T t = count) { }
```

The member function `f` of class `X` is defined in global scope; the notation `X::f` specifies that the function `f` is a member of class `X` and in the scope of class `X`. In the function definition, the parameter type `T` refers to the typedef member `T` declared in class `X` and the default argument `count` refers to the static data member `count` declared in class `X`. — *end example*]

6 A `static` local variable in a member function always refers to the same object, whether or not the member function is `inline`.

7 Member functions may be mentioned in `friend` declarations after their class has been defined.

8 Member functions of a local class shall be defined inline in their class definition, if they are defined at all.

9 [*Note:* a member function can be declared (but not defined) using a typedef for a function type. The resulting member function has exactly the same type as it would have if the function declarator were provided explicitly, see 8.3.5. For example,

```
typedef void fv(void);
typedef void fvc(void) const;
struct S {
  fv memfunc1;      // equivalent to: void memfunc1(void);
  void memfunc2();
  fvc memfunc3;     // equivalent to: void memfunc3(void) const;
};
fv  S::* pmfv1 = &S::memfunc1;
fv  S::* pmfv2 = &S::memfunc2;
fvc S::* pmfv3 = &S::memfunc3;
```

Also see 14.3. — *end note*]

9.3.1 Nonstatic member functions [class.mfct.non-static]

1 A *non-static* member function may be called for an object of its class type, or for an object of a class derived (Clause 10) from its class type, using the class member access syntax (5.2.5, 13.3.1.1). A non-static member function may also be called directly using the function call syntax (5.2.2, 13.3.1.1) from within the body of a member function of its class or of a class derived from its class.

2 If a non-static member function of a class `X` is called for an object that is not of type `X`, or of a type derived from `X`, the behavior is undefined.

3 When an *id-expression* (5.1) that is not part of a class member access syntax (5.2.5) and not used to form a pointer to member (5.3.1) is used in the body of a non-static member function of class `X`, if name lookup (3.4.1) resolves the name in the *id-expression* to a non-static non-type member of some class `C`, the *id-expression* is transformed into a class member access expression (5.2.5) using `(*this)` (9.3.2) as the *postfix-expression* to the left of the `.` operator. [*Note:* if `C` is not `X` or a base class of `X`, the class member access expression is ill-formed. — *end note*] Similarly during name lookup, when an *unqualified-id* (5.1) used in the definition of a member function for class `X` resolves to a `static` member, an enumerator or a nested type of class `X` or of a base class of `X`, the *unqualified-id* is transformed into a *qualified-id* (5.1) in which the *nested-name-specifier* names the class of the member function. [*Example:*

```
struct tnode {
  char tword[20];
  int count;
  tnode *left;
```

```
  tnode *right;
  void set(char*, tnode* l, tnode* r);
};

void tnode::set(char* w, tnode* l, tnode* r) {
  count = strlen(w)+1;
  if (sizeof(tword)<=count)
      perror("tnode string too long");
  strcpy(tword,w);
  left = l;
  right = r;
}

void f(tnode n1, tnode n2) {
  n1.set("abc",&n2,0);
  n2.set("def",0,0);
}
```

In the body of the member function `tnode::set`, the member names `tword`, `count`, `left`, and `right` refer to members of the object for which the function is called. Thus, in the call `n1.set("abc",&n2,0)`, `tword` refers to `n1.tword`, and in the call `n2.set("def",0,0)`, it refers to `n2.tword`. The functions `strlen`, `perror`, and `strcpy` are not members of the class `tnode` and should be declared elsewhere.[100] — *end example*]

4 A non-static member function may be declared `const`, `volatile`, or `const volatile`. These *cv-qualifiers* affect the type of the `this` pointer (9.3.2). They also affect the function type (8.3.5) of the member function; a member function declared `const` is a *const* member function, a member function declared `volatile` is a *volatile* member function and a member function declared `const volatile` is a *const volatile* member function. [*Example:*

```
struct X {
  void g() const;
  void h() const volatile;
};
```

`X::g` is a `const` member function and `X::h` is a `const volatile` member function. — *end example*]

5 A non-static member function may be declared with a *ref-qualifier* (8.3.5); see 13.3.1.

6 A non-static member function may be declared *virtual* (10.3) or *pure virtual* (10.4).

9.3.2 The `this` pointer [class.this]

1 In the body of a non-static (9.3) member function, the keyword `this` is an rvalue expression whose value is the address of the object for which the function is called. The type of `this` in a member function of a class `X` is `X*`. If the member function is declared `const`, the type of `this` is `const X*`, if the member function is declared `volatile`, the type of `this` is `volatile X*`, and if the member function is declared `const volatile`, the type of `this` is `const volatile X*`.

2 In a `const` member function, the object for which the function is called is accessed through a `const` access path; therefore, a `const` member function shall not modify the object and its non-static data members. [*Example:*

```
struct s {
  int a;
  int f() const;
```

100) See, for example, `<cstring>` (21.5).

```
  int g() { return a++; }
  int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The `a++` in the body of `s::h` is ill-formed because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function because `this` is a pointer to `const`; that is, `*this` has `const` type. — *end example*]

3 Similarly, `volatile` semantics (7.1.6.1) apply in `volatile` member functions when accessing the object and its non-static data members.

4 A *cv-qualified* member function can be called on an object-expression (5.2.5) only if the object-expression is as cv-qualified or less-cv-qualified than the member function. [*Example:*

```
void k(s& x, const s& y) {
  x.f();
  x.g();
  y.f();
  y.g();                          // error
}
```

The call `y.g()` is ill-formed because `y` is `const` and `s::g()` is a non-`const` member function, that is, `s::g()` is less-qualified than the object-expression y. — *end example*]

5 Constructors (12.1) and destructors (12.4) shall not be declared `const`, `volatile` or `const volatile`. [*Note:* However, these functions can be invoked to create and destroy objects with cv-qualified types, see (12.1) and (12.4). — *end note*]

9.4 Static members [class.static]

1 A data or function member of a class may be declared `static` in a class definition, in which case it is a *static member* of the class.

2 A `static` member `s` of class `X` may be referred to using the *qualified-id* expression `X::s`; it is not necessary to use the class member access syntax (5.2.5) to refer to a `static` member. A `static` member may be referred to using the class member access syntax, in which case the *object-expression* is evaluated. [*Example:*

```
struct process {
  static void reschedule();
};
process& g();

void f() {
  process::reschedule();          // OK: no object necessary
  g().reschedule();               // g() is called
}
```

— *end example*]

3 A `static` member may be referred to directly in the scope of its class or in the scope of a class derived (Clause 10) from its class; in this case, the `static` member is referred to as if a *qualified-id* expression was used, with the *nested-name-specifier* of the *qualified-id* naming the class scope from which the static member is referenced. [*Example:*

```
int g();
struct X {
  static int g();
};
struct Y : X {
  static int i;
};
int Y::i = g();                 // equivalent to Y::g();
```

— *end example*]

4 If an *unqualified-id* (5.1) is used in the definition of a `static` member following the member's *declarator-id*, and name lookup (3.4.1) finds that the *unqualified-id* refers to a `static` member, enumerator, or nested type of the member's class (or of a base class of the member's class), the *unqualified-id* is transformed into a *qualified-id* expression in which the *nested-name-specifier* names the class scope from which the member is referenced. [*Note:* See 5.1 for restrictions on the use of non-static data members and non-static member functions. — *end note*]

5 Static members obey the usual class member access rules (Clause 11). When used in the declaration of a class member, the `static` specifier shall only be used in the member declarations that appear within the *member-specification* of the class definition. [*Note:* it cannot be specified in member declarations that appear in namespace scope. — *end note*]

9.4.1 Static member functions [class.static.mfct]

1 [*Note:* the rules described in 9.3 apply to `static` member functions. — *end note*]

2 [*Note:* a `static` member function does not have a `this` pointer (9.3.2). — *end note*] A `static` member function shall not be `virtual`. There shall not be a `static` and a non-static member function with the same name and the same parameter types (13.1). A `static` member function shall not be declared `const`, `volatile`, or `const volatile`.

9.4.2 Static data members [class.static.data]

1 A `static` data member is not part of the subobjects of a class. If a `static` data member is declared `thread_local` there is one copy of the member per thread. If a `static` data member is not declared `thread_local` there is one copy of the data member that is shared by all the objects of the class.

2 The declaration of a `static` data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified `void`. The definition for a `static` data member shall appear in a namespace scope enclosing the member's class definition. In the definition at namespace scope, the name of the `static` data member shall be qualified by its class name using the `::` operator. The *initializer* expression in the definition of a `static` data member is in the scope of its class (3.3.6). [*Example:*

```
class process {
  static process* run_chain;
  static process* running;
};

process* process::running = get_main();
process* process::run_chain = running;
```

The `static` data member `run_chain` of class `process` is defined in global scope; the notation `process::run_chain` specifies that the member `run_chain` is a member of class `process` and in the scope of class `process`. In the `static` data member definition, the *initializer* expression refers to the `static` data member `running` of class `process`. — *end example*]

[*Note:* once the `static` data member has been defined, it exists even if no objects of its class have been created. [*Example:* in the example above, `run_chain` and `running` exist even if no objects of class `process` are created by the program. — *end example*] — *end note*]

3 If a `static` data member is of `const` effective literal type, its declaration in the class definition can specify a *constant-initializer brace-or-equal-initializer* with an *initializer-clause* that is an integral constant expression. A `static` data member of effective literal type can be declared in the class definition with the `constexpr` specifier; if so, its declaration shall specify a *constant-initializer brace-or-equal-initializer* with an *initializer-clause* that is an integral constant expression. In both these cases, the member may appear in integral constant expressions. The member shall still be defined in a namespace scope if it is used in the program and the namespace scope definition shall not contain an *initializer*.

4 There shall be exactly one definition of a `static` data member that is used in a program; no diagnostic is required; see 3.2. Unnamed classes and classes contained directly or indirectly within unnamed classes shall not contain `static` data members.

5 `Static` data members of a class in namespace scope have external linkage (3.5). A local class shall not have `static` data members.

6 `Static` data members are initialized and destroyed exactly like non-local objects (3.6.2, 3.6.3).

7 A `static` data member shall not be `mutable` (7.1.1).

9.5 Unions [class.union]

1 In a union, at most one of the data members can be active at any time, that is, the value of at most one of the data members can be stored in a union at any time. [*Note:* one special guarantee is made in order to simplify the use of unions: If a standard-layout union contains several standard-layout structs that share a common initial sequence (9.2), and if an object of this standard-layout union type contains one of the standard-layout structs, it is permitted to inspect the common initial sequence of any of standard-layout struct members; see 9.2. — *end note*] The size of a union is sufficient to contain the largest of its data members. Each data member is allocated as if it were the sole member of a struct. A union can have member functions (including constructors and destructors), but not virtual (10.3) functions. A union shall not have base classes. A union shall not be used as a base class. If a union contains a non-`static` data member of reference type the program is ill-formed. [*Note:* if any non-static data member of a union has a non-trivial default constructor (12.1), copy constructor (12.8), copy assignment operator (12.8), or destructor (12.4), the corresponding member function of the union must be user-declared user-provided or it will be implicitly deleted (8.4) for the union. — *end note*]

[*Example:* Consider the following union:

```
union U {
  int i;
  float f;
  std::string s;
};
```

Since `std::string` (21.2) declares non-trivial versions of all of the special member functions, `U` will have an implicitly deleted default constructor, copy constructor, copy assignment operator, and destructor. To use `U`, some or all of these member functions must be user-declared.

Consider an object `u` of a `union` type `U` having non-static data members `m` of type `M` and `n` of type `N`. If `M` has a non-trivial destructor and `N` has a non-trivial constructor (for instance, if they declare or inherit virtual functions), the active member of `u` can be safely switched from `m` to `n` using the destructor and placement new operator as follows:

```
u.m.~M();
new (&u.n) N;
```

— *end example*]

2 A union of the form

union *{ member-specification } ;*

is called an anonymous union; it defines an unnamed object of unnamed type. The *member-specification* of an anonymous union shall only define non-static data members. [*Note:* nested types and functions cannot be declared within an anonymous union. — *end note*] The names of the members of an anonymous union shall be distinct from the names of any other entity in the scope in which the anonymous union is declared. For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared. [*Example:*

```
void f() {
  union { int a; char* p; };
  a = 1;
  p = "Jennifer";
}
```

3 Here `a` and `p` are used like ordinary (nonmember) variables, but since they are union members they have the same address. — *end example*]

Anonymous unions declared in a named namespace or in the global namespace shall be declared `static`. Anonymous unions declared at block scope shall be declared with any storage class allowed for a block-scope variable, or with no storage class. A storage class is not allowed in a declaration of an anonymous union in a class scope. An anonymous union shall not have `private` or `protected` members (Clause 11). An anonymous union shall not have function members.

4 A union for which objects or pointers are declared is not an anonymous union. [*Example:*

```
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1;                         // error
ptr->aa = 1;                    // OK
```

The assignment to plain `aa` is ill-formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object. — *end example*] [*Note:* Initialization of unions with no user-declared constructors is described in (8.5.1). — *end note*]

5 A *union-like class* is a union or a class that has an anonymous union as a direct member. A union-like class `X` has a set of *variant members*. If `X` is a union its variant members are the non-static data members; otherwise, its variant members are the non-static data members of all anonymous unions that are members of `X`.

9.6 Bit-fields [class.bit]

1 A *member-declarator* of the form

identifier$_{opt}$ *attribute-specifier*$_{opt}$ **:** *constant-expression*

specifies a bit-field; its length is set off from the bit-field name by a colon. The optional *attribute-specifier* appertains to the entity being declared. The bit-field attribute is not part of the type of the class member. The *constant-expression* shall be an integral constant expression with a value greater than or equal to zero. The value of the integral constant expression may be larger than the number of bits in the object representation (3.9) of the bit-field's type; in such cases the extra bits are used as padding bits and do not

participate in the value representation (3.9) of the bit-field. Allocation of bit-fields within a class object is implementation-defined. Alignment of bit-fields is implementation-defined. Bit-fields are packed into some addressable allocation unit. [*Note:* bit-fields straddle allocation units on some machines and not on others. Bit-fields are assigned right-to-left on some machines, left-to-right on others. — *end note*]

2 A declaration for a bit-field that omits the *identifier* declares an *unnamed* bit-field. Unnamed bit-fields are not members and cannot be initialized. [*Note:* an unnamed bit-field is useful for padding to conform to externally-imposed layouts. — *end note*] As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary. Only when declaring an unnamed bit-field may the value of the *constant-expression* be equal to zero.

3 A bit-field shall not be a static member. A bit-field shall have integral or enumeration type (3.9.1). It is implementation-defined whether a plain (neither explicitly signed nor unsigned) `char`, `short`, `int` or `long` bit-field is signed or unsigned. A `bool` value can successfully be stored in a bit-field of any nonzero size. The address-of operator `&` shall not be applied to a bit-field, so there are no pointers to bit-fields. A non-const reference shall not be bound to a bit-field (8.5.3). [*Note:* if the initializer for a reference of type `const T&` is an lvalue that refers to a bit-field, the reference is bound to a temporary initialized to hold the value of the bit-field; the reference is not bound to the bit-field directly. See 8.5.3. — *end note*]

4 If the value `true` or `false` is stored into a bit-field of type `bool` of any size (including a one bit bit-field), the original `bool` value and the value of the bit-field shall compare equal. If the value of an enumerator is stored into a bit-field of the same enumeration type and the number of bits in the bit-field is large enough to hold all the values of that enumeration type (7.2), the original enumerator value and the value of the bit-field shall compare equal. [*Example:*

```
enum BOOL { FALSE=0, TRUE=1 };
struct A {
  BOOL b:1;
};
A a;
void f() {
  a.b = TRUE;
  if (a.b == TRUE)              // yields true
    { /* ... */ }
}
```

— *end example*]

9.7 Nested class declarations [class.nest]

1 A class can be declared within another class. A class declared within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. [*Note:* see 5.1 for restrictions on the use of non-static data members and non-static member functions. — *end note*]

[*Example:*

```
int x;
int y;

struct enclose {
  int x;
  static int s;

  struct inner {
    void f(int i) {
```

```
      int a = sizeof(x);        // OK: operand of sizeof is an unevaluated operand
      x = i;                    // error: assign to enclose::x
      s = i;                    // OK: assign to enclose::s
      ::x = i;                  // OK: assign to global x
      y = i;                    // OK: assign to global y
    }
    void g(enclose* p, int i) {
      p->x = i;                 // OK: assign to enclose::x
    }
  };
};

inner* p = 0;                   // error: inner not in scope
```

— *end example*]

2 Member functions and static data members of a nested class can be defined in a namespace scope enclosing the definition of their class. [*Example:*

```
struct enclose {
  struct inner {
    static int x;
    void f(int i);
  };
};

int enclose::inner::x = 1;

void enclose::inner::f(int i) { /* ... */ }
```

— *end example*]

3 If class X is defined in a namespace scope, a nested class Y may be declared in class X and later defined in the definition of class X or be later defined in a namespace scope enclosing the definition of class X. [*Example:*

```
class E {
  class I1;                     // forward declaration of nested class
  class I2;
  class I1 { };                 // definition of nested class
};
class E::I2 { };                // definition of nested class
```

— *end example*]

4 Like a member function, a friend function (11.4) defined within a nested class is in the lexical scope of that class; it obeys the same rules for name binding as a static member function of that class (9.4), but it has no special access rights to members of an enclosing class.

9.8 Local class declarations [class.local]

1 A class can be declared within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope, and has the same access to names outside the function as does the enclosing function. Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope. [*Example:*

```
int x;
void f() {
  static int s ;
  int x;
  extern int g();

  struct local {
    int g() { return x; }       // error: x has automatic storage duration
    int h() { return s; }       // OK
    int k() { return ::x; }     // OK
    int l() { return g(); }     // OK
  };
}

local* p = 0;                   // error: local not in scope
```

— *end example*]

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (Clause 11). Member functions of a local class shall be defined within their class definition, if they are defined at all.

3 If class `X` is a local class a nested class `Y` may be declared in class `X` and later defined in the definition of class `X` or be later defined in the same scope as the definition of class `X`. A class nested within a local class is a local class.

4 A local class shall not have static data members.

9.9 Nested type names [class.nested.type]

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. [*Example:*

```
struct X {
  typedef int I;
  class Y { /* ... */ };
  I a;
};

I b;                            // error
Y c;                            // error
X::Y d;                         // OK
X::I e;                         // OK
```

— *end example*]

10 Derived classes [class.derived]

1 A list of base classes can be specified in a class definition using the notation:

base-clause:
: *base-specifier-list*

base-specifier-list:
base-specifier ...$_{opt}$
base-specifier-list , *base-specifier* ...$_{opt}$

base-specifier:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* *attribute-specifier*$_{opt}$
`virtual` *access-specifier*$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* *attribute-specifier*$_{opt}$
access-specifier `virtual`$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name* *attribute-specifier*$_{opt}$

access-specifier:
`private`
`protected`
`public`

The optional *attribute-specifier* appertains to the *base-specifier*.

2 The *class-name* in a *base-specifier* shall not be an incompletely defined class (Clause class); this class is called a *direct base class* for the class being defined. During the lookup for a base class name, non-type names are ignored (3.3.10). If the name found is not a *class-name*, the program is ill-formed. A class `B` is a base class of a class `D` if it is a direct base class of `D` or a direct base class of one of `D`'s base classes. A class is an *indirect* base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. [*Note:* see Clause class.access for the meaning of *access-specifier*. — *end note*] Unless redeclared in the derived class, members of a base class are also considered to be members of the derived class. The base class members are said to be *inherited* by the derived class. Inherited members can be referred to in expressions in the same manner as other members of the derived class, unless their names are hidden or ambiguous (10.2). [*Note:* the scope resolution operator `::` (5.1) can be used to refer to a direct or indirect base member explicitly. This allows access to a name that has been redeclared in the derived class. A derived class can itself serve as a base class subject to access control; see 11.2. A pointer to a derived class can be implicitly converted to a pointer to an accessible unambiguous base class (4.10). An lvalue of a derived class type can be bound to a reference to an accessible unambiguous base class (8.5.3). — *end note*]

3 The *base-specifier-list* specifies the type of the *base class subobjects* contained in an object of the derived class type. [*Example:*

```
struct Base {
  int a, b, c;
};

struct Derived : Base {
  int b;
};

struct Derived2 : Derived {
  int c;
};
```

Here, an object of class `Derived2` will have a subobject of class `Derived` which in turn will have a subobject of class `Base`. — *end example*]

4 A *base-specifier* followed by an ellipsis is a pack expansion (14.5.3).

5 The order in which the base class subobjects are allocated in the most derived object (1.8) is unspecified. [*Note:* a derived class and its base class subobjects can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of subobjects is often referred to as a "subobject lattice."

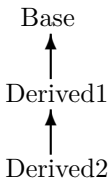

Figure 1 — Directed acyclic graph

6 The arrows need not have a physical representation in memory. — *end note*]

7 [*Note:* initialization of objects representing base classes can be specified in constructors; see 12.6.2. — *end note*]

8 [*Note:* A base class subobject might have a layout (3.7) different from the layout of a most derived object of the same type. A base class subobject might have a polymorphic behavior (12.7) different from the polymorphic behavior of a most derived object of the same type. A base class subobject may be of zero size (Clause class); however, two subobjects that have the same class type and that belong to the same most derived object must not be allocated at the same address (5.10). — *end note*]

10.1 Multiple base classes [class.mi]

1 A class can be derived from any number of base classes. [*Note:* the use of more than one direct base class is often called multiple inheritance. — *end note*] [*Example:*

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

— *end example*]

2 [*Note:* the order of derivation is not significant except as specified by the semantics of initialization by constructor (12.6.2), cleanup (12.4), and storage layout (9.2, class.access.spec). — *end note*]

3 A class shall not be specified as a direct base class of a derived class more than once. [*Note:* a class can be an indirect base class more than once and can be a direct and an indirect base class. There are limited things that can be done with such a class. The non-static data members and member functions of the direct base class cannot be referred to in the scope of the derived class. However, the static members, enumerations and types can be unambiguously referred to. — *end note*] [*Example:*

```
class X { /* ... */ };
class Y : public X, public X { /* ... */ };              // ill-formed
```

```
class L { public: int next;  /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { void f(); /* ... */ };    // well-formed
class D : public A, public L { void f(); /* ... */ };    // well-formed
```

— *end example*]

4 A base class specifier that does not contain the keyword `virtual`, specifies a *non-virtual* base class. A base class specifier that contains the keyword `virtual`, specifies a *virtual* base class. For each distinct occurrence of a non-virtual base class in the class lattice of the most derived class, the most derived object (1.8) shall contain a corresponding distinct base class subobject of that type. For each distinct base class that is specified virtual, the most derived object shall contain a single base class subobject of that type. [*Example:* for an object of class type `C`, each distinct occurrence of a (non-virtual) base class `L` in the class lattice of `C` corresponds one-to-one with a distinct `L` subobject within the object of type `C`. Given the class `C` defined above, an object of class `C` will have two subobjects of class `L` as shown below.

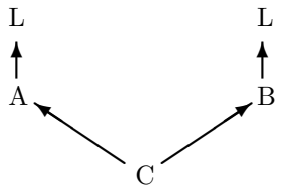

Figure 2 — Non-virtual base

5 In such lattices, explicit qualification can be used to specify which subobject is meant. The body of function `C::f` could refer to the member `next` of each `L` subobject:

```
void C::f() { A::next = B::next; }      // well-formed
```

Without the `A::` or `B::` qualifiers, the definition of `C::f` above would be ill-formed because of ambiguity (10.2).

6 For another example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

for an object `c` of class type `C`, a single subobject of type `V` is shared by every base subobject of `c` that has a `virtual` base class of type `V`. Given the class `C` defined above, an object of class `C` will have one subobject of class `V`, as shown below.

7 A class can have both virtual and non-virtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

For an object of class `AA`, all `virtual` occurrences of base class `B` in the class lattice of `AA` correspond to a single `B` subobject within the object of type `AA`, and every other occurrence of a (non-virtual) base class `B`

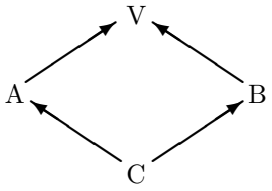

Figure 3 — Virtual base

in the class lattice of `AA` corresponds one-to-one with a distinct `B` subobject within the object of type `AA`. Given the class `AA` defined above, class `AA` has two subobjects of class `B`: `Z`'s `B` and the virtual `B` shared by `X` and `Y`, as shown below.

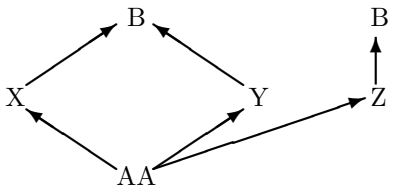

Figure 4 — Virtual and non-virtual base

— *end example*]

10.2 Member name lookup [class.member.lookup]

1 Member name lookup determines the meaning of a name (*id-expression*) in a class scope (3.3.6). Name lookup can result in an *ambiguity*, in which case the program is ill-formed. For an *id-expression*, name lookup begins in the class scope of `this`; for a *qualified-id*, name lookup begins in the scope of the *nested-name-specifier*. Name lookup takes place before access control (3.4, Clause class.access).

2 The following steps define the result of name lookup for a member name `f` in a class scope `C`.

3 The *lookup set* for `f` in `C`, called $S(f, C)$, consists of two component sets: the *declaration set*, a set of members named `f`; and the *subobject set*, a set of subobjects where declarations of these members (possibly including *using-declaration*s) were found. In the declaration set, *using-declaration*s are replaced by the members they designate, and type declarations (including injected-class-names) are replaced by the types they designate. $S(f, C)$ is calculated as follows:

4 If `C` contains a declaration of the name `f`, the declaration set contains every declaration of `f` declared in `C` that satisfies the requirements of the language construct in which the lookup occurs. [*Note:* Looking up a name in an *elaborated-type-specifier* (3.4.4) or *base-specifier* (Clause 10), for instance, ignores all non-type declarations, while looking up a name in a *nested-name-specifier* (3.4.3) ignores function, object, and enumerator declarations. As another example, looking up a name in a *using-declaration* (7.3.3) includes the declaration of a class or enumeration that would ordinarily be hidden by another declaration of that name in the same scope. — *end note*] If the resulting declaration set is not empty, the subobject set contains `C` itself, and calculation is complete.

5 Otherwise (i.e., `C` does not contain a declaration of `f` or the resulting declaration set is empty), $S(f, C)$ is initially empty. If `C` has base classes, calculate the lookup set for `f` in each direct base class subobject B_i, and merge each such lookup set $S(f, B_i)$ in turn into $S(f, C)$.

6 The following steps define the result of merging lookup set $S(f, B_i)$ into the intermediate $S(f, C)$:

— If each of the subobject members of $S(f, B_i)$ is a base class subobject of at least one of the subobject members of $S(f, C)$, or if $S(f, B_i)$ is empty, $S(f, C)$ is unchanged and the merge is complete. Conversely, if each of the subobject members of $S(f, C)$ is a base class subobject of at least one of the subobject members of $S(f, B_i)$, or if $S(f, C)$ is empty, the new $S(f, C)$ is a copy of $S(f, B_i)$.

— Otherwise, if the declaration sets of $S(f, B_i)$ and $S(f, C)$ differ, the merge is ambiguous: the new $S(f, C)$ is a lookup set with an invalid declaration set and the union of the subobject sets. In subsequent merges, an invalid declaration set is considered different from any other.

— Otherwise, the new $S(f, C)$ is a lookup set with the shared set of declarations and the union of the subobject sets.

7 The result of name lookup for `f` in `C` is the declaration set of $S(f, C)$. If it is an invalid set, the program is ill-formed. [*Example:*

```
struct A { int x; };                    // S(x,A) = { { A::x }, { A } }
struct B { float x; };                  // S(x,B) = { { B::x }, { B } }
struct C: public A, public B { };       // S(x,C) = { invalid, { A in C, B in C } }
struct D: public virtual C { };         // S(x,D) = S(x,C)
struct E: public virtual C { char x; }; // S(x,E) = { { E::x }, { E } }
struct F: public D, public E { };       // S(x,F) = S(x,E)
int main() {
  F f;
  f.x = 0;                              // OK, lookup finds E::x
}
```

$S(x, F)$ is unambiguous because the `A` and `B` base subobjects of `D` are also base subobjects of `E`, so $S(x, D)$ is discarded in the first merge step. — *end example*]

8 If the name of an overloaded function is unambiguously found, overloading resolution (13.3) also takes place before access control. Ambiguities can often be resolved by qualifying a name with its class name. [*Example:*

```
struct A {
  int f();
};

struct B {
  int f();
};

struct C : A, B {
  int f() { return A::f() + B::f(); }
};
```

— *end example*]

9 [*Note:* A static member, a nested type or an enumerator defined in a base class `T` can unambiguously be found even if an object has more than one base class subobject of type `T`. Two base class subobjects share the non-static member subobjects of their common virtual base classes. — *end note*] [*Example:*

```
struct V {
  int v;
};
struct A {
```

```
  int a;
  static int   s;
  enum { e };
};
struct B : A, virtual V { };
struct C : A, virtual V { };
struct D : B, C { };

void f(D* pd) {
  pd->v++;          // OK: only one v (virtual)
  pd->s++;          // OK: only one s (static)
  int i = pd->e;    // OK: only one e (enumerator)
  pd->a++;          // error, ambiguous: two as in D
}
```

— *end example*]

10 [*Note:* When virtual base classes are used, a hidden declaration can be reached along a path through the subobject lattice that does not pass through the hiding declaration. This is not an ambiguity. The identical use with non-virtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. — *end note*] [*Example:*

```
struct V { int f();  int x; };
struct W { int g();  int y; };
struct B : virtual V, W {
  int f();  int x;
  int g();  int y;
};
struct C : virtual V, W { };

struct D : B, C { void glorp(); };
```

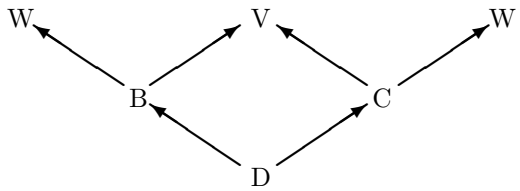

Figure 5 — Name lookup

— *end example*]

11 [*Note:* The names declared in V and the left-hand instance of W are hidden by those in B, but the names declared in the right-hand instance of W are not hidden at all. — *end note*]

```
void D::glorp() {
  x++;              // OK: B::x hides V::x
  f();              // OK: B::f() hides V::f()
  y++;              // error: B::y and C's W::y
  g();              // error: B::g() and C's W::g()
}
```

— *end example*]

12 An explicit or implicit conversion from a pointer to or an lvalue of a derived class to a pointer or reference to one of its base classes shall unambiguously refer to a unique object representing the base class. [*Example:*

```
struct V { };
struct A { };
struct B : A, virtual V { };
struct C : A, virtual V { };
struct D : B, C { };

void g() {
  D d;
  B* pb = &d;
  A* pa = &d;        // error, ambiguous: C's A or B's A?
  V* pv = &d;        // OK: only one V subobject
}
```

— *end example*]

13 [*Note:* Even if the result of name lookup is unambiguous, use of a name found in multiple subobjects might still be ambiguous (4.11, 5.2.5, 11.2). — *end note*] [*Example:*

```
struct B1 {
  void f();
  static void f(int);
  int i;
};
struct B2 {
  void f(double);
};
struct I1: B1 { };
struct I2: B1 { };

struct D: I1, I2, B2 {
  using B1::f;
  using B2::f;
  void g() {
    f();                        // Ambiguous conversion of this
    f(0);                       // Unambiguous (static)
    f(0.0);                     // Unambiguous (only one B2)
    int B1::* mpB1 = &D::i;     // Unambiguous
    int D::* mpD = &D::i;       // Ambiguous conversion
  }
};
```

— *end example*]

10.3 Virtual functions [class.virtual]

1 Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a *polymorphic class.*

2 If a virtual member function `vf` is declared in a class `Base` and in a class `Derived`, derived directly or indirectly from `Base`, a member function `vf` with the same name, parameter-type-list (8.3.5), cv-qualification, and ref-qualifier (or absence of same) as `Base::vf` is declared, then `Derived::vf` is also virtual (whether or not

it is so declared) and it *overrides*[101] `Base::vf`. For convenience we say that any virtual function overrides itself. Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function. The rules for member lookup (10.2) are used to determine the final overrider for a virtual function in the scope of a derived class but ignoring names introduced by *using-declaration*s. [*Example:*

```
struct A {
  virtual void f();
};
struct B : virtual A {
  virtual void f();
};
struct C : B , virtual A {
  using A::f;
};

void foo() {
  C c;
  c.f();             // calls B::f, the final overrider
  c.C::f();          // calls A::f because of the using-declaration
}
```

— *end example*]

3 [*Note:* a virtual member function does not have to be visible to be overridden, for example,

```
struct B {
  virtual void f();
};
struct D : B {
  void f(int);
};
struct D2 : D {
  void f();
};
```

the function `f(int)` in class `D` hides the virtual function `f()` in its base class `B`; `D::f(int)` is not a virtual function. However, `f()` declared in class `D2` has the same name and the same parameter list as `B::f()`, and therefore is a virtual function that overrides the function `B::f()` even though `B::f()` is not visible in class `D2`. — *end note*]

4 Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual; see class.dtor and class.free.

5 The return type of an overriding function shall be either identical to the return type of the overridden function or *covariant* with the classes of the functions. If a function `D::f` overrides a function `B::f`, the return types of the functions are covariant if they satisfy the following criteria:

— both are pointers to classes or references to classes[102]

— the class in the return type of `B::f` is the same class as the class in the return type of `D::f`, or is an unambiguous and accessible direct or indirect base class of the class in the return type of `D::f`

101) A function with the same name but a different parameter list (Clause over) as a virtual function is not necessarily virtual and does not override. The use of the `virtual` specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (Clause class.access) is not considered in determining overriding.
102) Multi-level pointers to classes or references to multi-level pointers to classes are not allowed.

— both pointers or references have the same cv-qualification and the class type in the return type of `D::f` has the same cv-qualification as or less cv-qualification than the class type in the return type of `B::f`.

6 If the return type of `D::f` differs from the return type of `B::f`, the class type in the return type of `D::f` shall be complete at the point of declaration of `D::f` or shall be the class type `D`. When the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function (5.2.2). [*Example:*

```
class B { };
class D : private B { friend class Derived; };
struct Base {
  virtual void vf1();
  virtual void vf2();
  virtual void vf3();
  virtual B*   vf4();
  virtual B*   vf5();
  void f();
};

struct No_good : public Base {
  D*  vf4();        // error: B (base class of D) inaccessible
};

class A;
struct Derived : public Base {
    void vf1();     // virtual and overrides Base::vf1()
    void vf2(int);  // not virtual, hides Base::vf2()
    char vf3();     // error: invalid difference in return type only
    D*  vf4();      // OK: returns pointer to derived class
    A*  vf5();      // error: returns pointer to incomplete class
    void f();
};

void g() {
  Derived d;
  Base* bp = &d;                // standard conversion:
                                // Derived* to Base*
  bp->vf1();                    // calls Derived::vf1()
  bp->vf2();                    // calls Base::vf2()
  bp->f();                      // calls Base::f() (not virtual)
  B*  p = bp->vf4();            // calls Derived::pf() and converts the
                                // result to B*
  Derived*  dp = &d;
  D*  q = dp->vf4();            // calls Derived::pf() and does not
                                // convert the result to B*
  dp->vf2();                    // ill-formed: argument mismatch
}
```

— *end example*]

7 [*Note:* the interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a non-virtual member function depends only on the type of the pointer or reference denoting that object (the static type) (5.2.2). — *end note*]

8 [*Note:* the `virtual` specifier implies membership, so a virtual function cannot be a nonmember (7.1.2) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific

object for determining which function to invoke. A virtual function declared in one class can be declared a `friend` in another class. — *end note*]

9 A virtual function declared in a class shall be defined, or declared pure (10.4) in that class, or both; but no diagnostic is required (3.2).

10 [*Example:* here are some uses of virtual functions with multiple base classes:

```
struct A {
  virtual void f();
};

struct B1 : A {                 // note non-virtual derivation
  void f();
};

struct B2 : A {
  void f();
};

struct D : B1, B2 {             // D has two separate A subobjects
};

void foo() {
  D  d;
// A* ap = &d; // would be ill-formed: ambiguous
  B1*  b1p = &d;
  A*   ap = b1p;
  D*   dp = &d;
  ap->f();                      // calls D::B1::f
  dp->f();                      // ill-formed: ambiguous
}
```

In class `D` above there are two occurrences of class `A` and hence two occurrences of the virtual member function `A::f`. The final overrider of `B1::A::f` is `B1::f` and the final overrider of `B2::A::f` is `B2::f`.

11 The following example shows a function that does not have a unique final overrider:

```
struct A {
  virtual void f();
};

struct VB1 : virtual A {        // note virtual derivation
  void f();
};

struct VB2 : virtual A {
  void f();
};

struct Error : VB1, VB2 {       // ill-formed
};

struct Okay : VB1, VB2 {
  void f();
};
```

Both `VB1::f` and `VB2::f` override `A::f` but there is no overrider of both of them in class `Error`. This example is therefore ill-formed. Class `Okay` is well formed, however, because `Okay::f` is a final overrider.

12 The following example uses the well-formed classes from above.

```
struct VB1a : virtual A {        // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe() {
  VB1a*  vb1ap = new Da;
  vb1ap->f();                    // calls VB2::f
}
```

— *end example*]

13 Explicit qualification with the scope operator (5.1) suppresses the virtual call mechanism. [*Example:*

```
class B { public: virtual void f(); };
class D : public B { public: void f(); };

void D::f() { /* ... */ B::f(); }
```

Here, the function call in `D::f` really does call `B::f` and not `D::f`. — *end example*]

14 A function with a deleted definition (8.4) shall not override a function that does not have a deleted definition. Likewise, a function that does not have a deleted definition shall not override a function with a deleted definition.

10.4 Abstract classes [class.abstract]

1 The abstract class mechanism supports the notion of a general concept, such as a `shape`, of which only more concrete variants, such as `circle` and `square`, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

2 An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it. A class is abstract if it has at least one *pure virtual function*. [*Note:* such a function might be inherited: see below. — *end note*] A virtual function is specified *pure* by using a *pure-specifier* (9.2) in the function declaration in the class definition. A pure virtual function need be defined only if called with, or as if with (12.4), the *qualified-id* syntax (5.1). [*Example:*

```
class point { /* ... */ };
class shape {                    // abstract class
  point center;
public:
  point where() { return center; }
  void move(point p) { center=p; draw(); }
  virtual void rotate(int) = 0; // pure virtual
  virtual void draw() = 0;       // pure virtual
};
```

— *end example*] [*Note:* a function declaration cannot provide both a *pure-specifier* and a definition — *end note*] [*Example:*

```
struct C {
  virtual void f() = 0 { };      // ill-formed
};
```

— *end example*]

3 An abstract class shall not be used as a parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class can be declared. [*Example:*

```
shape x;                        // error: object of abstract class
shape* p;                       // OK
shape f();                      // error
void g(shape);                  // error
shape& h(shape&);               // OK
```

— *end example*]

4 A class is abstract if it contains or inherits at least one pure virtual function for which the final overrider is pure virtual. [*Example:*

```
class ab_circle : public shape {
  int radius;
public:
  void rotate(int) { }
  // ab_circle::draw() is a pure virtual
};
```

Since `shape::draw()` is a pure virtual function `ab_circle::draw()` is a pure virtual by default. The alternative declaration,

```
class circle : public shape {
  int radius;
public:
  void rotate(int) { }
  void draw();                  // a definition is required somewhere
};
```

would make class `circle` nonabstract and a definition of `circle::draw()` must be provided. — *end example*]

5 [*Note:* an abstract class can be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure. — *end note*]

6 Member functions can be called from a constructor (or destructor) of an abstract class; the effect of making a virtual call (10.3) to a pure virtual function directly or indirectly for the object being created (or destroyed) from such a constructor (or destructor) is undefined.

11 Member access control [class.access]

1 A member of a class can be

— `private`; that is, its name can be used only by members and friends of the class in which it is declared.

— `protected`; that is, its name can be used only by members and friends of the class in which it is declared, by classes derived from that class, and by their friends (see 11.5).

— `public`; that is, its name can be used anywhere without access restriction.

2 A member of a class can also access all the names to which the class has access. A local class of a member function may access the same names that the member function itself may access.[103]

3 Members of a class defined with the keyword `class` are `private` by default. Members of a class defined with the keywords `struct` or `union` are `public` by default. [*Example:*

```
class X {
  int a;            // X::a is private by default
};

struct S {
  int a;            // S::a is public by default
};
```

— *end example*]

4 Access control is applied uniformly to all names, whether the names are referred to from declarations or expressions. [*Note:* access control applies to names nominated by `friend` declarations (11.4) and *using-declarations* (7.3.3). — *end note*] In the case of overloaded function names, access control is applied to the function selected by overload resolution. [*Note:* because access control applies to names, if access control is applied to a typedef name, only the accessibility of the typedef name itself is considered. The accessibility of the entity referred to by the typedef is not considered. For example,

```
class A {
  class B { };
public:
  typedef B BB;
};

void f() {
  A::BB x;          // OK, typedef name A::BB is public
  A::B y;           // access error, A::B is private
}
```

— *end note*]

5 It should be noted that it is *access* to members and base classes that is controlled, not their *visibility*. Names of members are still visible, and implicit conversions to base classes are still considered, when those members and base classes are inaccessible. The interpretation of a given construct is established without regard to

103) Access permissions are thus transitive and cumulative to nested and local classes.

access control. If the interpretation established makes use of inaccessible member names or base classes, the construct is ill-formed.

6 All access controls in Clause 11 affect the ability to access a class member name from a particular scope. For purposes of access control, the *base-specifier*s of a class and the definitions of class members that appear outside of the class definition are considered to be within the scope of that class. In particular, access controls apply as usual to member names accessed as part of a function return type, even though it is not possible to determine the access privileges of that use without first parsing the rest of the function declarator. Similarly, access control for implicit calls to the constructors, the conversion functions, or the destructor called to create and destroy a static data member is performed as if these calls appeared in the scope of the member's class. [*Example:*

```
class A {
  typedef int I;     // private member
  I f();
  friend I g(I);
  static I x;
protected:
    struct B { };
};

A::I A::f() { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I A::x = 0;

struct D: A::B, A { };
```

7 Here, all the uses of `A::I` are well-formed because `A::f` and `A::x` are members of class `A` and `g` is a friend of class `A`. This implies, for example, that access checking on the first use of `A::I` must be deferred until it is determined that this use of `A::I` is as the return type of a member of class `A`. Similarly, the use of `A::B` as a *base-specifier* is well-formed because `D` is derived from `A`, so checking of *base-specifier*s must be deferred until the entire *base-specifier-list* has been seen. — *end example*]

8 The names in a default argument expression (8.3.6) are bound at the point of declaration, and access is checked at that point rather than at any points of use of the default argument expression. Access checking for default arguments in function templates and in member functions of class templates is performed as described in 14.7.1.

9 The names in a default *template-argument* (14.1) have their access checked in the context in which they appear rather than at any points of use of the default *template-argument*. [*Example:*

```
class B { };
template <class T> class C {
protected:
  typedef T TT;
};

template <class U, class V = typename U::TT>
class D : public U { };

D <C<B> >* d;        // access error, C::TT is protected
```

— *end example*]

11.1 Access specifiers [class.access.spec]

1 Member declarations can be labeled by an *access-specifier* (Clause 10):

access-specifier : *member-specification*$_{opt}$

An *access-specifier* specifies the access rules for members following it until the end of the class or until another *access-specifier* is encountered. [*Example:*

```
class X {
  int a;            // X::a is private by default: class used
public:
  int b;            // X::b is public
  int c;            // X::c is public
};
```

— *end example*]

2 Any number of access specifiers is allowed and no particular order is required. [*Example:*

```
struct S {
  int a;            // S::a is public by default: struct used
protected:
  int b;            // S::b is protected
private:
  int c;            // S::c is private
public:
  int d;            // S::d is public
};
```

— *end example*]

3 [*Note:* the effect of access control on the order of allocation of data members is described in 9.2. — *end note*]

4 When a member is redeclared within its class definition, the access specified at its redeclaration shall be the same as at its initial declaration. [*Example:*

```
struct S {
  class A;
private:
  class A { };      // error: cannot change access
};
```

— *end example*]

5 [*Note:* In a derived class, the lookup of a base class name will find the injected-class-name instead of the name of the base class in the scope in which it was declared. The injected-class-name might be less accessible than the name of the base class in the scope in which it was declared. — *end note*]

[*Example:*

```
class A { };
class B : private A { };
class C : public B {
  A *p;             // error: injected-class-name A is inaccessible
  ::A *q;           // OK
};
```

— *end example*]

11.2 Accessibility of base classes and base class members [class.access.base]

1 If a class is declared to be a base class (Clause 10) for another class using the `public` access specifier, the `public` members of the base class are accessible as `public` members of the derived class and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `protected` access specifier, the `public` and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `private` access specifier, the `public` and `protected` members of the base class are accessible as `private` members of the derived class[104].

2 In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is defined with the *class-key* `struct` and `private` is assumed when the class is defined with the *class-key* `class`. [*Example:*

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };      // B private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };     // B public by default
class D7 : protected B { /* ... */ };
struct D8 : protected B { /* ... */ };
```

Here `B` is a public base of `D2`, `D4`, and `D6`, a private base of `D1`, `D3`, and `D5`, and a protected base of `D7` and `D8`. — *end example*]

3 [*Note:* A member of a private base class might be inaccessible as an inherited member name, but accessible directly. Because of the rules on pointer conversions (4.10) and explicit casts (5.4), a conversion from a pointer to a derived class to a pointer to an inaccessible base class might be ill-formed if an implicit conversion is used, but well-formed if an explicit cast is used. For example,

```
class B {
public:
  int mi;                       // non-static member
  static int si;                // static member
};
class D : private B {
};
class DD : public D {
  void f();
};

void DD::f() {
  mi = 3;                       // error: mi is private in D
  si = 3;                       // error: si is private in D
  ::B  b;
  b.mi = 3;                     // OK (b.mi is different from this->mi)
  b.si = 3;                     // OK (b.si is different from this->si)
  ::B::si = 3;                  // OK
  ::B* bp1 = this;              // error: B is a private base class
  ::B* bp2 = (::B*)this;        // OK with cast
```

104) As specified previously in Clause 11, private members of a base class remain inaccessible even to derived classes unless `friend` declarations within the base class definition are used to grant access explicitly.

```
  bp2->mi = 3;                    // OK: access through a pointer to B.
}
```

— *end note*]

4 A base class `B` of `N` is *accessible* at R, if

— an invented public member of `B` would be a public member of `N`, or

— R occurs in a member or friend of class `N`, and an invented public member of `B` would be a private or protected member of `N`, or

— R occurs in a member or friend of a class `P` derived from `N`, and an invented public member of `B` would be a private or protected member of `P`, or

— there exists a class `S` such that `B` is a base class of `S` accessible at R and `S` is a base class of `N` accessible at R.

[*Example:*

```
class B {
public:
  int m;
};

class S: private B {
  friend class N;
};

class N: private S {
  void f() {
    B* p = this;     // OK because class S satisfies the fourth condition
                     // above: B is a base class of N accessible in f() because
                     // B is an accessible base class of S and S is an accessible
                     // base class of N.
  }
};
```

— *end example*]

5 If a base class is accessible, one can implicitly convert a pointer to a derived class to a pointer to that base class (4.10, 4.11). [*Note:* it follows that members and friends of a class `X` can implicitly convert an `X*` to a pointer to a private or protected immediate base class of `X`. — *end note*] The access to a member is affected by the class in which the member is named. This naming class is the class in which the member name was looked up and found. [*Note:* this class can be explicit, e.g., when a *qualified-id* is used, or implicit, e.g., when a class member access operator (5.2.5) is used (including cases where an implicit "`this->`" is added). If both a class member access operator and a *qualified-id* are used to name the member (as in `p->T::m`), the class naming the member is the class named by the *nested-name-specifier* of the *qualified-id* (that is, `T`). — *end note*] A member `m` is accessible at the point R when named in class `N` if

— `m` as a member of `N` is public, or

— `m` as a member of N is private, and R occurs in a member or friend of class `N`, or

— `m` as a member of `N` is protected, and R occurs in a member or friend of class `N`, or in a member or friend of a class `P` derived from `N`, where `m` as a member of `P` is public, private, or protected, or

— there exists a base class `B` of `N` that is accessible at R, and `m` is accessible at R when named in class `B`. [*Example:*

```
class B;
class A {
private:
  int i;
  friend void f(B*);
};
class B : public A { };
void f(B* p) {
  p->i = 1;          // OK: B* can be implicitly converted to A*,
                     // and f has access to i in A
}
```

— *end example*]

6 If a class member access operator, including an implicit "`this->`," is used to access a non-static data member or non-static member function, the reference is ill-formed if the left operand (considered as a pointer in the "`.`" operator case) cannot be implicitly converted to a pointer to the naming class of the right operand. [*Note:* this requirement is in addition to the requirement that the member be accessible as named. — *end note*]

11.3 Access declarations [class.access.dcl]

1 The access of a member of a base class can be changed in the derived class by mentioning its *qualified-id* in the derived class definition. Such mention is called an *access declaration*. The effect of an access declaration *qualified-id* `;` is defined to be equivalent to the declaration `using` *qualified-id* `;`.[105]

[*Example:*

```
class A {
public:
  int z;
  int z1;
};

class B : public A {
  int a;
public:
  int b, c;
  int bf();
protected:
  int x;
  int y;
};

class D : private B {
  int d;
public:
  B::c;              // adjust access to B::c
```

105) Access declarations are deprecated; member *using-declaration*s (7.3.3) provide a better means of doing the same things. In earlier versions of the C++ language, access declarations were more limited; they were generalized and made equivalent to *using-declaration*s in the interest of simplicity. Programmers are encouraged to use *using-declaration*s, rather than the new capabilities of access declarations, in new code.

```
  B::z;             // adjust access to A::z
  A::z1;            // adjust access to A::z1
  int e;
  int df();
protected:
  B::x;             // adjust access to B::x
  int g;
};

class X : public D {
  int xf();
};

int ef(D&);
int ff(X&);
```

The external function `ef` can use only the names `c`, `z`, `z1`, `e`, and `df`. Being a member of `D`, the function `df` can use the names `b`, `c`, `z`, `z1`, `bf`, `x`, `y`, `d`, `e`, `df`, and `g`, but not `a`. Being a member of `B`, the function `bf` can use the members `a`, `b`, `c`, `z`, `z1`, `bf`, `x`, and `y`. The function `xf` can use the public and protected names from `D`, that is, `c`, `z`, `z1`, `e`, and `df` (public), and `x`, and `g` (protected). Thus the external function `ff` has access only to `c`, `z`, `z1`, `e`, and `df`. If `D` were a protected or private base class of `X`, `xf` would have the same privileges as before, but `ff` would have no access at all. — *end example*]

11.4 Friends [class.friend]

1 A friend of a class is a function or class that is given permission to use the private and protected member names from the class. A class specifies its friends, if any, by way of friend declarations. Such declarations give special access rights to the friends, but they do not make the nominated friends members of the befriending class. [*Example:* the following example illustrates the differences between members and friends:

```
class X {
  int a;
  friend void friend_set(X*, int);
public:
  void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }

void f() {
  X obj;
  friend_set(&obj,10);
  obj.member_set(10);
}
```

— *end example*]

2 Declaring a class to be a friend implies that the names of private and protected members from the class granting friendship can be accessed in the *base-specifier*s and member declarations of the befriended class. [*Example:*

```
class A {
  class B { };
  friend class X;
};
```

```
struct X : A::B {    // OK: A::B accessible to friend
  A::B mx;           // OK: A::B accessible to member of friend
  class Y {
    A::B my;         // OK: A::B accessible to nested member of friend
  };
};
```

— *end example*] A class shall not be defined in a friend declaration. [*Example:*

```
class X {
  enum { a=100 };
  friend class Y;
};

class Y {
  int v[X::a];       // OK, Y is a friend of X
};

class Z {
  int v[X::a];       // error: X::a is private
};
```

— *end example*]

3 A `friend` declaration that does not declare a function shall have one of the following forms:

`friend` *elaborated-type-specifier* `;`
`friend` *simple-type-specifier* `;`
`friend` *typename-specifier* `;`

[*Note:* a `friend` declaration may be the *declaration* in a *template-declaration* (Clause 14, 14.5.4). — *end note*] If the type specifier in a `friend` declaration designates a (possibly cv-qualified) class type, that class is declared as a friend; otherwise, the `friend` declaration is ignored. [*Example:*

```
class C;
typedef C Ct;

class X1 {
  friend C;          // OK: class C is a friend
};

class X2 {
  friend Ct;         // OK: class C is a friend
  friend D;          // error: no type-name D in scope
  friend class D;    // OK: elaborated-type-specifier declares new class
};

template <typename T> class R {
  friend T;
};

R<C> rc;             // class C is a friend of R<C>
R<int> Ri;           // OK: "friend int;" is ignored
```

— *end example*]

4 A function first declared in a friend declaration has external linkage (3.5). Otherwise, the function retains its previous linkage (7.1.1).

5 When a `friend` declaration refers to an overloaded name or operator, only the function specified by the parameter types becomes a friend. A member function of a class `X` can be a friend of a class `Y`. [*Example:*

```
class Y {
  friend char* X::foo(int);
  friend X::X(char);          // constructors can be friends
  friend X::~X();             // destructors can be friends
};
```

— *end example*]

6 A function can be defined in a friend declaration of a class if and only if the class is a non-local class (9.8), the function name is unqualified, and the function has namespace scope. [*Example:*

```
class M {
  friend void f() { }         // definition of global f, a friend of M,
                              // not the definition of a member function
};
```

— *end example*]

7 Such a function is implicitly `inline`. A `friend` function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not (3.4.1).

8 No *storage-class-specifier* shall appear in the *decl-specifier-seq* of a friend declaration.

9 A name nominated by a friend declaration shall be accessible in the scope of the class containing the friend declaration. The meaning of the friend declaration is the same whether the friend declaration appears in the `private`, `protected` or `public` (9.2) portion of the class *member-specification*.

10 Friendship is neither inherited nor transitive. [*Example:*

```
class A {
  friend class B;
  int a;
};

class B {
  friend class C;
};

class C  {
  void f(A* p) {
    p->a++;          // error: C is not a friend of A
                     // despite being a friend of a friend
  }
};

class D : public B  {
  void f(A* p) {
    p->a++;          // error: D is not a friend of A
                     // despite being derived from a friend
  }
};
```

— *end example*]

11 If a friend declaration appears in a local class (9.8) and the name specified is an unqualified name, a prior declaration is looked up without considering scopes that are outside the innermost enclosing non-class scope. For a friend function declaration, if there is no prior declaration, the program is ill-formed. For a friend class declaration, if there is no prior declaration, the class that is specified belongs to the innermost enclosing non-class scope, but if it is subsequently referenced, its name is not found by name lookup until a matching declaration is provided in the innermost enclosing nonclass scope. [*Example:*

```
class X;
void a();
void f() {
  class Y;
  extern void b();
  class A {
  friend class X;   // OK, but X is a local class, not ::X
  friend class Y;   // OK
  friend class Z;   // OK, introduces local class Z
  friend void a();  // error, ::a is not considered
  friend void b();  // OK
  friend void c();  // error
  };
  X *px;            // OK, but ::X is found
  Z *pz;            // error, no Z is found
}
```

— *end example*]

11.5 Protected member access [class.protected]

1 An additional access check beyond those described earlier in Clause 11 is applied when a non-static data member or non-static member function is a protected member of its naming class (11.2)[106] As described earlier, access to a protected member is granted because the reference occurs in a friend or member of some class `C`. If the access is to form a pointer to member (5.3.1), the *nested-name-specifier* shall name `C` or a class derived from `C`. All other accesses involve a (possibly implicit) object expression (5.2.5). In this case, the class of the object expression shall be `C` or a class derived from `C`. [*Example:*

```
class B {
protected:
  int i;
  static int j;
};

class D1 : public B {
};

class D2 : public B {
  friend void fr(B*,D1*,D2*);
  void mem(B*,D1*);
};

void fr(B* pb, D1* p1, D2* p2) {
  pb->i = 1;                    // ill-formed
  p1->i = 2;                    // ill-formed
```

106) This additional check does not apply to other members, *e.g.* static data members or enumerator member constants.

```
  p2->i = 3;                    // OK (access through a D2)
  p2->B::i = 4;                 // OK (access through a D2, even though
                                // naming class is B)
  int B::* pmi_B = &B::i;       // ill-formed
  int B::* pmi_B2 = &D2::i;     // OK (type of &D2::i is int B::*)
  B::j = 5;                     // OK (because refers to static member)
  D2::j =6;                     // OK (because refers to static member)
}

void D2::mem(B* pb, D1* p1) {
  pb->i = 1;                    // ill-formed
  p1->i = 2;                    // ill-formed
  i = 3;                        // OK (access through this)
  B::i = 4;                     // OK (access through this, qualification ignored)
  int B::* pmi_B = &B::i;       // ill-formed
  int B::* pmi_B2 = &D2::i;     // OK
  j = 5;                        // OK (because j refers to static member)
  B::j = 6;                     // OK (because B::j refers to static member)
}

void g(B* pb, D1* p1, D2* p2) {
  pb->i = 1;                    // ill-formed
  p1->i = 2;                    // ill-formed
  p2->i = 3;                    // ill-formed
}
```

— *end example*]

11.6 Access to virtual functions [class.access.virt]

1 The access rules (Clause 11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. [*Example:*

```
class B {
public:
  virtual int f();
};

class D : public B {
private:
  int f();
};

void f() {
  D d;
  B* pb = &d;
  D* pd = &d;

  pb->f();                      // OK: B::f() is public,
                                // D::f() is invoked
  pd->f();                      // error: D::f() is private
}
```

— *end example*]

2 Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (`B*` in the example above). The access of the member function in the class in which it was defined (`D` in the example above) is in general not known.

11.7 Multiple access [class.paths]

1 If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. [*Example:*

```
class W { public: void f(); };
class A : private virtual W { };
class B : public virtual W { };
class C : public A, public B {
  void f() { W::f(); }          // OK
};
```

2 Since `W::f()` is available to `C::f()` along the public path through `B`, access is allowed. — *end example*]

11.8 Nested classes [class.access.nest]

1 A nested class is a member and as such has the same access rights as any other member. The members of an enclosing class have no special access to members of a nested class; the usual access rules (Clause 11) shall be obeyed. [*Example:*

```
class E {
  int x;
  class B { };

  class I {
    B b;                        // OK: E::I can access E::B
    int y;
    void f(E* p, int i) {
      p->x = i;                 // OK: E::I can access E::x
    }
  };

  int g(I* p) {
    return p->y;                // error: I::y is private
  }
};
```

— *end example*]

12 Special member functions [special]

1 The default constructor (12.1), copy constructor and copy assignment operator (12.8), and destructor (12.4) are *special member functions*. [*Note:* The implementation will implicitly declare these member functions for some class types when the program does not explicitly declare them. The implementation will implicitly define them if they are used. See 12.1, 12.4 and 12.8. —*end note*] Programs shall not define implicitly-declared special member functions. Programs may explicitly refer to implicitly-declared special member functions. [*Example:* a program may explicitly call, take the address of or form a pointer to member to an implicitly-declared special member function.

```
struct A { };                   // implicitly-declared A::operator=
struct B : A {
  B& operator=(const B &);
};
B& B::operator=(const B& s) {
  this->A::operator=(s);        // well-formed
  return *this;
}
```

—*end example*]

2 [*Note:* the special member functions affect the way objects of class type are created, copied, and destroyed, and how values can be converted to values of other types. Often such special member functions are called implicitly. —*end note*]

3 Special member functions obey the usual access rules (Clause 11). [*Example:* declaring a constructor `protected` ensures that only derived classes and friends can create objects using it. —*end example*]

12.1 Constructors [class.ctor]

1 Constructors do not have names. A special declarator syntax using an optional sequence of *function-specifier*s (7.1.2) followed by the constructor's class name followed by a parameter list is used to declare or define the constructor. In such a declaration, optional parentheses around the constructor class name are ignored. [*Example:*

```
struct S {
  S();              // declares the constructor
};

S::S() { }          // defines the constructor
```

—*end example*]

2 A constructor is used to initialize objects of its class type. Because constructors do not have names, they are never found during name lookup; however an explicit type conversion using the functional notation (5.2.3) will cause a constructor to be called to initialize an object. [*Note:* for initialization of objects of class type see 12.6. —*end note*]

3 A *typedef-name* shall not be used as the *class-name* in the *declarator-id* for a constructor declaration.

4 A constructor shall not be `virtual` (10.3) or `static` (9.4). A constructor can be invoked for a `const`, `volatile` or `const volatile` object. A constructor shall not be declared `const`, `volatile`, or `const`

`volatile` (9.3.2). `const` and `volatile` semantics (7.1.6.1) are not applied on an object under construction. They come into effect when the constructor for the most derived object (1.8) ends. A constructor shall not be declared with a *ref-qualifier*.

5 A *default* constructor for a class `X` is a constructor of class `X` that can be called without an argument. If there is no user-declared constructor for class `X`, a constructor having no parameters is implicitly declared. An implicitly-declared default constructor is an `inline public` member of its class. A default constructor is *trivial* if it is not user-provided (8.4) and if:

- its class has no virtual functions (10.3) and no virtual base classes (10.1), and
- no non-static data member of its class has a *brace-or-equal-initializer*, and
- all the direct base classes of its class have trivial default constructors, and
- for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

An implicitly-declared default constructor for class `X` is defined as deleted if:

- `X` is a union-like class that has a variant member with a non-trivial default constructor,
- any non-static data member is of reference type,
- any non-static data member of const-qualified type (or array thereof) does not have a user-provided default constructor, or
- any non-static data member or direct or virtual base class has class type `M` (or array thereof) and `M` has no default constructor, or if overload resolution (13.3) as applied to `M`'s default constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared default constructor.

Otherwise, the default constructor is *non-trivial*.

6 A non-user-provided default constructor for a class is *implicitly defined* when it is used (3.2) to create an object of its class type (1.8). If the implicitly-defined default constructor is explicitly defaulted but the corresponding implicit declaration would have been deleted, the program is ill-formed. The implicitly-defined or explicitly-defaulted default constructor performs the set of initializations of the class that would be performed by a user-written default constructor for that class with no *ctor-initializer* (12.6.2) and an empty *compound-statement*. If that user-written default constructor would be ill-formed, the program is ill-formed. If that user-written default constructor would satisfy the requirements of a constexpr constructor (7.1.5), the implicitly-defined default constructor is constexpr. Before the non-user-provided default constructor for a class is implicitly defined, all the non-user-provided default constructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared default constructor has an *exception-specification* (15.4). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

7 Default constructors are called implicitly to create class objects of static, thread, or automatic storage duration (3.7.1, 3.7.2, 3.7.3) defined without an initializer (8.5), are called to create class objects of dynamic storage duration (3.7.4) created by a *new-expression* in which the *new-initializer* is omitted (5.3.4), or are called when the explicit type conversion syntax (5.2.3) is used. A program is ill-formed if the default constructor for an object is implicitly used and the constructor is not accessible (Clause 11).

8 [*Note:* 12.6.2 describes the order in which constructors for base classes and non-static data members are called and describes how arguments can be specified for the calls to these constructors. — *end note*]

9 A copy constructor (12.8) is used to copy objects of class type.

10 No return type (not even `void`) shall be specified for a constructor. A `return` statement in the body of a constructor shall not specify a return value. The address of a constructor shall not be taken.

11 A functional notation type conversion (5.2.3) can be used to create new objects of its type. [*Note:* The syntax looks like an explicit call of the constructor. — *end note*] [*Example:*

```
complex zz = complex(1,2.3);
cprint( complex(7.8,1.2) );
```

— *end example*]

12 An object created in this way is unnamed. [*Note:* 12.2 describes the lifetime of temporary objects. — *end note*] [*Note:* explicit constructor calls do not yield lvalues, see 3.10. — *end note*]

13 [*Note:* some language constructs have special semantics when used during construction; see 12.6.2 and 12.7. — *end note*]

14 During the construction of a `const` object, if the value of the object or any of its subobjects is accessed through an lvalue that is not obtained, directly or indirectly, from the constructor's `this` pointer, the value of the object or subobject thus obtained is unspecified. [*Example:*

```
struct C;
void no_opt(C*);

struct C {
  int c;
  C() : c(0) { no_opt(this); }
};

const C cobj;

void no_opt(C* cptr) {
  int i = cobj.c * 100;          // value of cobj.c is unspecified
  cptr->c = 1;
  cout << cobj.c * 100           // value of cobj.c is unspecified
       << '\n';
}
```

— *end example*]

12.2 Temporary objects [class.temporary]

1 Temporaries of class type are created in various contexts: binding an rvalue to a reference (8.5.3), returning an rvalue (6.6.3), a conversion that creates an rvalue (4.1, 5.2.9, 5.2.11, 5.4), throwing an exception (15.1), entering a *handler* (15.3), and in some initializations (8.5). [*Note:* the lifetime of exception objects is described in 15.1. — *end note*] Even when the creation of the temporary object is avoided (12.8), all the semantic restrictions shall be respected as if the temporary object had been created. [*Example:* even if the copy constructor is not called, all the semantic restrictions, such as accessibility (Clause 11), shall be satisfied. — *end example*]

[*Example:*

```
class X {
public:
  X(int);
  X(const X&);
  ~X();
```

```
};

X f(X);

void g() {
  X a(1);
  X b = f(X(2));
  a = f(a);
}
```

2 Here, an implementation might use a temporary in which to construct `X(2)` before passing it to `f()` using `X`'s copy-constructor; alternatively, `X(2)` might be constructed in the space used to hold the argument. Also, a temporary might be used to hold the result of `f(X(2))` before copying it to `b` using `X`'s copy-constructor; alternatively, `f()`'s result might be constructed in `b`. On the other hand, the expression `a=f(a)` requires a temporary for the result of `f(a)`, which is then assigned to `a`. — *end example*]

3 When an implementation introduces a temporary object of a class that has a non-trivial constructor (12.1, 12.8), it shall ensure that a constructor is called for the temporary object. Similarly, the destructor shall be called for a temporary with a non-trivial destructor (12.4). Temporary objects are destroyed as the last step in evaluating the full-expression (1.9) that (lexically) contains the point where they were created. This is true even if that evaluation ends in throwing an exception. The value computations and side effects of destroying a temporary object are associated only with the full-expression, not with any specific subexpression.

4 There are two contexts in which temporaries are destroyed at a different point than the end of the full-expression. The first context is when a default constructor is called to initialize an element of an array. If the constructor has one or more default arguments, the destruction of every temporary created in a default argument expression is sequenced before the construction of the next array element, if any.

5 The second context is when a reference is bound to a temporary. The temporary to which the reference is bound or the temporary that is the complete object of a subobject to which the reference is bound persists for the lifetime of the reference except as specified below. A temporary bound to a reference member in a constructor's ctor-initializer (12.6.2) persists until the constructor exits. A temporary bound to a reference parameter in a function call (5.2.2) persists until the completion of the full-expression containing the call. A temporary bound to the returned value in a function return statement (6.6.3) persists until the function exits. A temporary bound to a reference in a *new-initializer* (5.3.4) persists until the completion of the full-expression containing the *new-initializer*. [*Example:*

```
struct S { int mi; const std::pair<int,int>& mp; };
S a { 1, {2,3} };
S* p = new S{ 1, {2,3} };   // Creates dangling reference
```

— *end example*] [*Note:* This may introduce a dangling reference, and implementations are encouraged to issue a warning in such a case. — *end note*] The destruction of a temporary whose lifetime is not extended by being bound to a reference is sequenced before the destruction of every temporary which is constructed earlier in the same full-expression. If the lifetime of two or more temporaries to which references are bound ends at the same point, these temporaries are destroyed at that point in the reverse order of the completion of their construction. In addition, the destruction of temporaries bound to references shall take into account the ordering of destruction of objects with static, thread, or automatic storage duration (3.7.1, 3.7.2, 3.7.3); that is, if `obj1` is an object with the same storage duration as the temporary and created before the temporary is created the temporary shall be destroyed before `obj1` is destroyed; if `obj2` is an object with the same storage duration as the temporary and created after the temporary is created the temporary shall be destroyed after `obj2` is destroyed. [*Example:*

```
struct S {
```

```
  S();
  S(int);
  friend S operator+(const S&, const S&);
  ~S();
};
S obj1;
const S& cr = S(16)+S(23);
S obj2;
```

the expression `C(16)+C(23)` creates three temporaries. A first temporary `T1` to hold the result of the expression `C(16)`, a second temporary `T2` to hold the result of the expression `C(23)`, and a third temporary `T3` to hold the result of the addition of these two expressions. The temporary `T3` is then bound to the reference `cr`. It is unspecified whether `T1` or `T2` is created first. On an implementation where `T1` is created before `T2`, it is guaranteed that `T2` is destroyed before `T1`. The temporaries `T1` and `T2` are bound to the reference parameters of `operator+`; these temporaries are destroyed at the end of the full-expression containing the call to `operator+`. The temporary `T3` bound to the reference `cr` is destroyed at the end of `cr`'s lifetime, that is, at the end of the program. In addition, the order in which `T3` is destroyed takes into account the destruction order of other objects with static storage duration. That is, because `obj1` is constructed before `T3`, and `T3` is constructed before `obj2`, it is guaranteed that `obj2` is destroyed before `T3`, and that `T3` is destroyed before `obj1`. — *end example*]

12.3 Conversions [class.conv]

1 Type conversions of class objects can be specified by constructors and by conversion functions. These conversions are called *user-defined conversions* and are used for implicit type conversions (Clause 4), for initialization (8.5), and for explicit type conversions (5.4, 5.2.9).

2 User-defined conversions are applied only where they are unambiguous (10.2, 12.3.2). Conversions obey the access control rules (Clause 11). Access control is applied after ambiguity resolution (3.4).

3 [*Note:* See 13.3 for a discussion of the use of conversions in function calls as well as examples below. — *end note*]

4 At most one user-defined conversion (constructor or conversion function) is implicitly applied to a single value.

[*Example:*

```
struct X {
  operator int();
};

struct Y {
  operator X();
};

Y a;
int b = a;          // error
                    // a.operator X().operator int() not tried
int c = X(a);       // OK: a.operator X().operator int()
```

— *end example*]

5 User-defined conversions are used implicitly only if they are unambiguous. A conversion function in a derived class does not hide a conversion function in a base class unless the two functions convert to the same

type. Function overload resolution (13.3.3) selects the best conversion function to perform the conversion. [*Example:*

```
struct X {
  operator int();
};

struct Y : X {
    operator char();
};

void f(Y& a) {
  if (a) {          // ill-formed:
                    //
```

3 A non-explicit copy-constructor (12.8) is a converting constructor. An implicitly-declared copy constructor is not an explicit constructor; it may be called for implicit type conversions.

12.3.2 Conversion functions [class.conv.fct]

1 A member function of a class `X` having no parameters, or an associated function of a concept whose sole parameter is of type `X`, with a name of the form

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq attribute-specifier$_{opt}$ *conversion-declarator*$_{opt}$

conversion-declarator:
ptr-operator conversion-declarator$_{opt}$

specifies a conversion from `X` to the type specified by the *conversion-type-id*. Such functions are called conversion functions. No return type can be specified. If a conversion function is a member function, the type of the conversion function (8.3.5) is "function taking no parameter returning *conversion-type-id*"; if a conversion function is an associated function, the type of the conversion function is "function taking a parameter of type `X` returning *conversion-type-id*". A conversion function is never used to convert a (possibly cv-qualified) object to the (possibly cv-qualified) same object type (or a reference to it), to a (possibly cv-qualified) base class of that type (or a reference to it), or to (possibly cv-qualified) void.[107]

[*Example:*

```
struct X {
  operator int();
};

void f(X a) {
  int i = int(a);
  i = (int)a;
  i = a;
}
```

In all three cases the value assigned will be converted by `X::operator int()`. — *end example*]

2 A conversion function may be explicit (7.1.2), in which case it is only considered as a user-defined conversion for direct-initialization (8.5). Otherwise, user-defined conversions are not restricted to use in assignments and initializations. [*Example:*

```
class Y { };
struct Z {
  explicit operator Y() const;
};

void h(Z z) {
  Y y1(z);          // OK: direct-initialization
  Y y2 = z;         // ill-formed: copy-initialization
  Y y3 = (Y)z;      // OK: cast notation
}

void g(X a, X b) {
  int i = (a) ? 1+a : 0;
```

107) Even though never directly called to perform a conversion, such conversion functions can be declared and can potentially be reached through a call to a virtual conversion function in a base class

```
  int j = (a&&b) ? a+b : i;
  if (a) {
  }
}
```

— *end example*]

3 The *conversion-type-id* shall not represent a function type nor an array type. The *conversion-type-id* in a *conversion-function-id* is the longest possible sequence of *conversion-declarator*s. [*Note:* this prevents ambiguities between the declarator operator * and its expression counterparts. [*Example:*

```
&ac.operator int*i; // syntax error:
                    // parsed as: &(ac.operator int *)i
                    // not as: &(ac.operator int)*i
```

The * is the pointer declarator and not the multiplication operator. — *end example*] — *end note*]

4 Conversion functions are inherited.

5 Conversion functions can be virtual.

6 Conversion functions cannot be declared `static`.

12.4 Destructors [class.dtor]

1 A special declarator syntax using an optional *function-specifier* (7.1.2) followed by ~ followed by the destructor's class name followed by an empty parameter list is used to declare the destructor in a class definition. In such a declaration, the ~ followed by the destructor's class name can be enclosed in optional parentheses; such parentheses are ignored. A *typedef-name* shall not be used as the *class-name* following the ~ in the declarator for a destructor declaration.

2 A destructor is used to destroy objects of its class type. A destructor takes no parameters, and no return type can be specified for it (not even `void`). The address of a destructor shall not be taken. A destructor shall not be `static`. A destructor can be invoked for a `const`, `volatile` or `const volatile` object. A destructor shall not be declared `const`, `volatile` or `const volatile` (9.3.2). `const` and `volatile` semantics (7.1.6.1) are not applied on an object under destruction. They stop being in effect when the destructor for the most derived object (1.8) starts. A destructor shall not be declared with a *ref-qualifier*.

3 If a class has no user-declared destructor, a destructor is declared implicitly. An implicitly-declared destructor is an `inline public` member of its class. If the class is a union-like class that has a variant member with a non-trivial destructor, an implicitly-declared destructor is defined as deleted (8.4). A destructor is *trivial* if it is not user-provided and if:

- the destructor is not `virtual`,
- all of the direct base classes of its class have trivial destructors, and
- for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

An implicitly-declared destructor for a class `X` is defined as deleted if:

- `X` is a union-like class that has a variant member with a non-trivial destructor,
- any of the non-static data members has class type `M` (or array thereof) and `M` has an deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor, or
- any direct or virtual base class has a deleted destructor or a destructor that is inaccessible from the implicitly-declared destructor.

Otherwise, the destructor is *non-trivial*.

4 A non-user-provided destructor is *implicitly defined* when it is used to destroy an object of its class type (3.7). A program is ill-formed if the class for which a destructor is implicitly defined or explicitly defaulted has: if the implicitly-defined destructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

— a non-static data member of class type (or array thereof) with an inaccessible destructor, or

— a base class with an inaccessible destructor.

Before the non-user-provided destructor for a class is implicitly defined, all the non-user-defined destructors for its base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared destructor has an *exception-specification* (15.4). An explictly defaulted definition has no implicit *exception-specification*. — *end note*]

5 After executing the body of the destructor and destroying any automatic objects allocated within the body, a destructor for class `X` calls the destructors for `X`'s direct non-variant members, the destructors for `X`'s direct base classes and, if `X` is the type of the most derived class (12.6.2), its destructor calls the destructors for `X`'s virtual base classes. All destructors are called as if they were referenced with a qualified name, that is, ignoring any possible virtual overriding destructors in more derived classes. Bases and members are destroyed in the reverse order of the completion of their constructor (see 12.6.2). A `return` statement (6.6.3) in a destructor might not directly return to the caller; before transferring control to the caller, the destructors for the members and bases are called. Destructors for elements of an array are called in reverse order of their construction (see 12.6).

6 A destructor can be declared `virtual` (10.3) or pure `virtual` (10.4); if any objects of that class or any derived class are created in the program, the destructor shall be defined. If a class has a base class with a virtual destructor, its destructor (whether user- or implicitly- declared) is virtual.

7 [*Note:* some language constructs have special semantics when used during destruction; see 12.7. — *end note*]

8 Destructors are invoked implicitly

— for constructed objects with static storage duration (3.7.1) at program termination (3.6.3),

— for constructed objects with thread storage duration (3.7.2) at thread exit,

— for constructed objects with automatic storage duration (3.7.3) when the block in which an object is created exits (6.7),

— for constructed temporary objects when the lifetime of a temporary object ends (12.2),

— for constructed objects allocated by a *new-expression* (5.3.4), through use of a *delete-expression* (5.3.5),

— in several situations due to the handling of exceptions (15.3).

A program is ill-formed if an object of class type or array thereof is declared and the destructor for the class is not accessible at the point of the declaration. Destructors can also be invoked explicitly.

9 At the point of definition of a virtual destructor (including an implicit definition (12.8)), the non-array deallocation function is looked up in the scope of the destructor's class (10.2), and, if no declaration is found, the function is looked up in the global scope. If the result of this lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function or a function with a deleted definition (8.4), the program is ill-formed. [*Note:* this assures that a deallocation function corresponding to the dynamic type of an object is available for the *delete-expression* (12.5). — *end note*]

10 In an explicit destructor call, the destructor name appears as a ~ followed by a *type-name* that names the destructor's class type. The invocation of a destructor is subject to the usual rules for member functions (9.3), that is, if the object is not of the destructor's class type and not of a class derived from the destructor's class type, the program has undefined behavior (except that invoking `delete` on a null pointer has no effect). [*Example:*

```
struct B {
  virtual ~B() { }
};
struct D : B {
  ~D() { }
};

D D_object;
typedef B B_alias;
B* B_ptr = &D_object;

void f() {
  D_object.B::~B();             // calls B's destructor
  B_ptr->~B();                  // calls D's destructor
  B_ptr->~B_alias();            // calls D's destructor
  B_ptr->B_alias::~B();         // calls B's destructor
  B_ptr->B_alias::~B_alias();   // calls B's destructor
}
```

— *end example*] [*Note:* an explicit destructor call must always be written using a member access operator (5.2.5) or a qualified-id (5.1); in particular, the *unary-expression* `~X()` in a member function is not an explicit destructor call (5.3.1). — *end note*]

11 [*Note:* explicit calls of destructors are rarely needed. One use of such calls is for objects placed at specific addresses using a *new-expression* with the placement option. Such use of explicit placement and destruction of objects can be necessary to cope with dedicated hardware resources and for writing memory management facilities. For example,

```
void* operator new(std::size_t, void* p) { return p; }
struct X {
  X(int);
  ~X();
};
void f(X* p);

void g() {                      // rare, specialized use:
  char* buf = new char[sizeof(X)];
  X* p = new(buf) X(222);       // use buf[] and initialize
  f(p);
  p->X::~X();                   // cleanup
}
```

— *end note*]

12 Once a destructor is invoked for an object, the object no longer exists; the behavior is undefined if the destructor is invoked for an object whose lifetime has ended (3.8). [*Example:* if the destructor for an automatic object is explicitly invoked, and the block is subsequently left in a manner that would ordinarily invoke implicit destruction of the object, the behavior is undefined. — *end example*]

13 [*Note:* the notation for explicit call of a destructor can be used for any scalar type name (5.2.4). Allowing this makes it possible to write code without having to know if a destructor exists for a given type. For example,

```
typedef int I;
I* p;
p->I::~I();
```

— *end note*]

12.5 Free store [class.free]

1 Any allocation function for a class `T` is a static member (even if not explicitly declared `static`).

2 [*Example:*

```
class Arena;
struct B {
  void* operator new(std::size_t, Arena*);
};
struct D1 : B {
};

Arena*  ap;
void foo(int i) {
  new (ap) D1;       // calls B::operator new(std::size_t, Arena*)
  new D1[i];         // calls ::operator new[](std::size_t)
  new D1;            // ill-formed: ::operator new(std::size_t) hidden
}
```

— *end example*]

3 When an object is deleted with a *delete-expression* (5.3.5), a *deallocation function* (`operator delete()` for non-array objects or `operator delete[]()` for arrays) is (implicitly) called to reclaim the storage occupied by the object (3.7.4.2).

4 If a *delete-expression* begins with a unary `::` operator, the deallocation function's name is looked up in global scope. Otherwise, if the *delete-expression* is used to deallocate a class object whose static type has a virtual destructor, the deallocation function is the one selected at the point of definition of the dynamic type's virtual destructor (12.4).[108] Otherwise, if the *delete-expression* is used to deallocate an object of class `T` or array thereof, the static and dynamic types of the object shall be identical and the deallocation function's name is looked up in the scope of `T`. If this lookup fails to find the name, the name is looked up in the global scope. If the result of the lookup is ambiguous or inaccessible, or if the lookup selects a placement deallocation function, the program is ill-formed.

5 When a *delete-expression* is executed, the selected deallocation function shall be called with the address of the block of storage to be reclaimed as its first argument and (if the two-parameter style is used) the size of the block as its second argument.[109]

6 Any deallocation function for a class `X` is a static member (even if not explicitly declared `static`). [*Example:*

108) A similar provision is not needed for the array version of `operator delete` because 5.3.5 requires that in this situation, the static type of the object to be deleted be the same as its dynamic type.

109) If the static type of the object to be deleted is different from the dynamic type and the destructor is not virtual the size might be incorrect, but that case is already undefined; see 5.3.5.

```
class X {
  void operator delete(void*);
  void operator delete[](void*, std::size_t);
};

class Y {
  void operator delete(void*, std::size_t);
  void operator delete[](void*);
};
```

— *end example*]

7 Since member allocation and deallocation functions are `static` they cannot be virtual. [*Note:* however, when the *cast-expression* of a *delete-expression* refers to an object of class type, because the deallocation function actually called is looked up in the scope of the class that is the dynamic type of the object, if the destructor is virtual, the effect is the same. For example,

```
struct B {
  virtual ~B();
  void operator delete(void*, std::size_t);
};

struct D : B {
  void operator delete(void*);
};

void f() {
  B* bp = new D;
  delete bp;        // 1: uses D::operator delete(void*)
}
```

Here, storage for the non-array object of class `D` is deallocated by `D::operator delete()`, due to the virtual destructor. — *end note*] [*Note:* virtual destructors have no effect on the deallocation function actually called when the *cast-expression* of a *delete-expression* refers to an array of objects of class type. For example,

```
struct B {
  virtual ~B();
  void operator delete[](void*, std::size_t);
};

struct D : B {
  void operator delete[](void*, std::size_t);
};

void f(int i) {
  D* dp = new D[i];
  delete [] dp;     // uses D::operator delete[](void*, std::size_t)
  B* bp = new D[i];
  delete[] bp;      // undefined behavior
}
```

— *end note*]

8 Access to the deallocation function is checked statically. Hence, even though a different one might actually be executed, the statically visible deallocation function is required to be accessible. [*Example:* for the call

on line //1 above, if `B::operator delete()` had been `private`, the delete expression would have been ill-formed. — *end example*]

12.6 Initialization [class.init]

1 When no initializer is specified for an object of (possibly cv-qualified) class type (or array thereof), or the initializer has the form `()`, the object is initialized as specified in 8.5.

2 An object of class type (or array thereof) can be explicitly initialized; see 12.6.1 and 12.6.2.

3 When an array of class objects is initialized (either explicitly or implicitly), the constructor shall be called for each element of the array, following the subscript order; see 8.3.4. [*Note:* destructors for the array elements are called in reverse order of their construction. — *end note*]

12.6.1 Explicit initialization [class.expl.init]

1 An object of class type can be initialized with a parenthesized *expression-list*, where the *expression-list* is construed as an argument list for a constructor that is called to initialize the object. Alternatively, a single *assignment-expression* can be specified as an *initializer* using the `=` form of initialization. Either direct-initialization semantics or copy-initialization semantics apply; see 8.5. [*Example:*

```
struct complex {
  complex();
  complex(double);
  complex(double,double);
};

complex sqrt(complex,complex);

complex a(1);                   // initialize by a call of
                                // complex(double)
complex b = a;                  // initialize by a copy of a
complex c = complex(1,2);       // construct complex(1,2)
                                // using complex(double,double)
                                // copy it into c
complex d = sqrt(b,c);          // call sqrt(complex,complex)
                                // and copy the result into d
complex e;                      // initialize by a call of
                                // complex()
complex f = 3;                  // construct complex(3) using
                                // complex(double)
                                // copy it into f
complex g = { 1, 2 };           // error: constructor is required
```

— *end example*] [*Note:* overloading of the assignment operator (13.5.3) has no effect on initialization. — *end note*]

2 An object of class type can also be initialized by a *braced-init-list*. List-initialization semantics apply; see 8.5 and 8.5.4. [*Example:*

```
complex v[6] = { 1, complex(1,2), complex(), 2 };
```

Here, `complex::complex(double)` is called for the initialization of `v[0]` and `v[3]`, `complex::complex(double, double)` is called for the initialization of `v[1]`, `complex::complex()` is called for the initialization `v[2]`, `v[4]`, and `v[5]`. For another example,

```
struct X {
  int i;
  float f;
  complex c;
} x = { 99, 88.8, 77.7 };
```

Here, `x.i` is initialized with 99, `x.f` is initialized with 88.8, and `complex::complex(double)` is called for the initialization of `x.c`. — *end example*] [*Note:* braces can be elided in the *initializer-list* for any aggregate, even if the aggregate has members of a class type with user-defined type conversions; see 8.5.1. — *end note*]

3 [*Note:* if `T` is a class type with no default constructor, any declaration of an object of type `T` (or array thereof) is ill-formed if no *initializer* is explicitly specified (see 12.6 and 8.5). — *end note*]

4 [*Note:* the order in which objects with static or thread storage duration are initialized is described in 3.6.2 and 6.7. — *end note*]

12.6.2 Initializing bases and members [class.base.init]

1 In the definition of a constructor for a class, initializers for direct and virtual base subobjects and non-static data members can be specified by a *ctor-initializer*, which has the form

ctor-initializer:
: *mem-initializer-list*

mem-initializer-list:
mem-initializer ...$_{opt}$
mem-initializer , *mem-initializer-list* ...$_{opt}$

mem-initializer:
mem-initializer-id (*expression-list*$_{opt}$)
mem-initializer-id braced-init-list

mem-initializer-id:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
identifier

2 Names in a *mem-initializer-id* are looked up in the scope of the constructor's class and, if not found in that scope, are looked up in the scope containing the constructor's definition. [*Note:* if the constructor's class contains a member with the same name as a direct or virtual base class of the class, a *mem-initializer-id* naming the member or base class and composed of a single identifier refers to the class member. A *mem-initializer-id* for the hidden base class may be specified using a qualified name. — *end note*] Unless the *mem-initializer-id* names the constructor's class, a non-static data member of the constructor's class or a direct or virtual base of that class, the *mem-initializer* is ill-formed. A *mem-initializer-list* can initialize a base class using any name that denotes that base class type. [*Example:*

```
struct A { A(); };
  typedef A global_A;
  struct B { };
  struct C: public A, public B { C(); };
  C::C(): global_A() { }        // mem-initializer for base A
```

— *end example*]

A *mem-initializer-list* can delegate to another constructor of the constructor's class using any name that denotes the constructor's class itself. If a *mem-initializer-id* designates the constructor's class, it shall be the only *mem-initializer*; the constructor is a *delegating constructor*, and the constructor selected by the *mem-initializer* is the *target constructor*. The *principal constructor* is the first constructor invoked in the construction of an object (that is, not a target constructor for that object's construction). The target constructor is selected by overload resolution. Once the target constructor returns, the body of the delegating

constructor is executed. If a constructor delegates to itself directly or indirectly, the program is ill-formed; no diagnostic is required. [*Example:*

```
struct C {
  C( int ) { }                  // 1: non-delegating constructor
  C(): C(42) { }                // 2: delegates to 1
  C( char c ) : C(42.0) { }     // 3: ill-formed due to recursion with 4
  C( double d ) : C('a') { }    // 4: ill-formed due to recursion with 3
};
```

— *end example*] If a *mem-initializer-id* is ambiguous because it designates both a direct non-virtual base class and an inherited virtual base class, the *mem-initializer* is ill-formed. [*Example:*

```
struct A { A(); };
struct B: public virtual A { };
struct C: public A, public B { C(); };
C::C(): A() { }                 // ill-formed: which A?
```

— *end example*] A *ctor-initializer* may initialize the member of an anonymous union that is a member of the constructor's class. If a *ctor-initializer* specifies more than one *mem-initializer* for the same member, for the same base class or for multiple members of the same union (including members of anonymous unions), the *ctor-initializer* is ill-formed.

3 The *expression-list* or *braced-init-list* in a *mem-initializer* is used to initialize the base class or non-static data member subobject denoted by the *mem-initializer-id* according to the initialization rules of 8.5 for direct-initialization.

[*Example:*

```
struct B1 { B1(int); /* ... */ };
struct B2 { B2(int); /* ... */ };
struct D : B1, B2 {
  D(int);
  B1 b;
  const int c;
};

D::D(int a) : B2(a+1), B1(a+2), c(a+3), b(a+4)
  { /* ... */ }
D d(10);
```

— *end example*] The initialization of each base and member constitutes a full-expression. Any expression in a *mem-initializer* is evaluated as part of the full-expression that performs the initialization.

4 If a given non-static data member or base class is not named by a *mem-initializer-id* (including the case where there is no *mem-initializer-list* because the constructor has no *ctor-initializer*), then

— if the entity is a non-static data member that has a *brace-or-equal-initializer*, the entity is initialized as specified in 8.5;

— otherwise, if the entity is a variant member (9.5), no initialization is performed;

— otherwise, the entity is default-initialized (8.5).

After the call to a constructor for class `X` has completed, if a member of `X` is neither initialized nor given a value during execution of the *compound-statement* of the body of the constructor, the member has indeterminate value. [*Example:*

```
struct A {
  A();
};

struct B {
  B(int);
};

struct C {
  C() { }                 // initializes members as follows:
  A a;                      // OK: calls A::A()
  const B b;                // error: B has no default constructor
  int i;                    // OK: i has indeterminate value
  int j = 5;                // OK: j has the value 5
};
```

— *end example*]

5 If a given non-static data member has both a *brace-or-equal-initializer* and a *mem-initializer*, the initialization specified by the *mem-initializer* is performed, and the non-static data member's *brace-or-equal-initializer* is ignored. [*Example:* Given

```
struct A {
  int i = /* some integer expression with side  effects  */ ;
  A(int arg) : i(arg) { }
  // ...
};
```

the `A(int)` constructor will simply initialize `i` to the value of `arg`, and the side effects in `i`'s *brace-or-equal-initializer* will not take place. — *end example*]

6 Initialization shall proceed in the following order:

— First, and only for the constructor of the most derived class as described below, virtual base classes shall be initialized in the order they appear on a depth-first left-to-right traversal of the directed acyclic graph of base classes, where "left-to-right" is the order of appearance of the base class names in the derived class *base-specifier-list*.

— Then, direct base classes shall be initialized in declaration order as they appear in the *base-specifier-list* (regardless of the order of the *mem-initializers*).

— Then, non-static data members shall be initialized in the order they were declared in the class definition (again regardless of the order of the *mem-initializers*).

— Finally, the *compound-statement* of the constructor body is executed.

[*Note:* the declaration order is mandated to ensure that base and member subobjects are destroyed in the reverse order of initialization. — *end note*]

7 All subobjects representing virtual base classes are initialized by the constructor of the most derived class (1.8). If the constructor of the most derived class does not specify a *mem-initializer* for a virtual base class `V`, then `V`'s default constructor is called to initialize the virtual base class subobject. If `V` does not have an accessible default constructor, the initialization is ill-formed. A *mem-initializer* naming a virtual base class shall be ignored during execution of the constructor of any class that is not the most derived class. [*Example:*

```
struct V {
```

```
  V();
  V(int);
};

struct A : virtual V {
  A();
  A(int);
};

struct B : virtual V {
  B();
  B(int);
};

struct C : A, B, virtual V {
  C();
  C(int);
};

A::A(int i) : V(i) { /* ... */ }
B::B(int i) { /* ... */ }
C::C(int i) { /* ... */ }

V v(1);             // use V(int)
A a(2);             // use V(int)
B b(3);             // use V()
C c(4);             // use V()
```

— *end example*]

8 Names in the *expression-list* of a *mem-initializer* are evaluated in the scope of the constructor for which the *mem-initializer* is specified. [*Example:*

```
class X {
  int a;
  int b;
  int i;
  int j;
public:
  const int& r;
  X(int i): r(a), b(i), i(i), j(this->i) { }
};
```

initializes `X::r` to refer to `X::a`, initializes `X::b` with the value of the constructor parameter `i`, initializes `X::i` with the value of the constructor parameter `i`, and initializes `X::j` with the value of `X::i`; this takes place each time an object of class `X` is created. — *end example*] [*Note:* because the *mem-initializer* are evaluated in the scope of the constructor, the `this` pointer can be used in the *expression-list* of a *mem-initializer* to refer to the object being initialized. — *end note*]

9 Member functions (including virtual member functions, 10.3) can be called for an object under construction. Similarly, an object under construction can be the operand of the `typeid` operator (5.2.8) or of a `dynamic_cast` (5.2.7). However, if these operations are performed in a *ctor-initializer* (or in a function called directly or indirectly from a *ctor-initializer*) before all the *mem-initializer*s for base classes have completed, the result of the operation is undefined. [*Example:*

```
class A {
```

```
public:
  A(int);
};

class B : public A {
  int j;
public:
  int f();
  B() : A(f()),      // undefined: calls member function
                     // but base A not yet initialized
  j(f()) { }         // well-defined: bases are all initialized
};

class C {
public:
  C(int);
};

class D : public B, C {
  int i;
public:
  D() : C(f()),      // undefined: calls member function
                     // but base C not yet initialized
  i(f()) { }         // well-defined: bases are all initialized
};
```

— *end example*]

10 [*Note:* 12.7 describes the result of virtual function calls, `typeid` and `dynamic_cast`s during construction for the well-defined cases; that is, describes the *polymorphic behavior* of an object under construction. — *end note*]

11 A *mem-initializer* followed by an ellipsis is a pack expansion (14.5.3) that initializes the base classes specified by a pack expansion in the *base-specifier-list* for the class. [*Example:*

```
template<class... Mixins>
class X : public Mixins... {
public:
  X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

— *end example*]

12.7 Construction and destruction [class.cdtor]

1 For an object with a non-trivial constructor, referring to any non-static member or base class of the object before the constructor begins execution results in undefined behavior. For an object with a non-trivial destructor, referring to any non-static member or base class of the object after the destructor finishes execution results in undefined behavior. [*Example:*

```
struct X { int i; };
struct Y : X { Y(); };                  // non-trivial
struct A { int a; };
struct B : public A { int j; Y y; };    // non-trivial

extern B bobj;
```

```
B* pb = &bobj;                          // OK
int* p1 = &bobj.a;                      // undefined, refers to base class member
int* p2 = &bobj.y.i;                    // undefined, refers to member's member

A* pa = &bobj;                          // undefined, upcast to a base class type
B bobj;                                 // definition of bobj

extern X xobj;
int* p3 = &xobj.i;                      //OK, X is a trivial class
X xobj;
```

2 For another example,

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
  int *p;
  X x;
  Y() : p(&x.j) {   // undefined, x is not yet constructed
    }
};
```

— *end example*]

3 To explicitly or implicitly convert a pointer (an lvalue) referring to an object of class `X` to a pointer (reference) to a direct or indirect base class `B` of `X`, the construction of `X` and the construction of all of its direct or indirect bases that directly or indirectly derive from `B` shall have started and the destruction of these classes shall not have completed, otherwise the conversion results in undefined behavior. To form a pointer to (or access the value of) a direct non-static member of an object `obj`, the construction of `obj` shall have started and its destruction shall not have completed, otherwise the computation of the pointer value (or accessing the member value) results in undefined behavior. [*Example:*

```
struct A { };
struct B : virtual A { };
struct C : B { };
struct D : virtual A { D(A*); };
struct X { X(A*); };

struct E : C, D, X {
  E() : D(this),    // undefined: upcast from E* to A*
                    // might use path E* → D* → A*
                    // but D is not constructed
                    // D((C*)this), // defined:
                    // E* → C* defined because E() has started
                    // and C* → A* defined because
                    // C fully constructed
  X(this) {         // defined: upon construction of X,
                    // C/B/D/A sublattice is fully constructed
  }
};
```

— *end example*]

4 Member functions, including virtual functions (10.3), can be called during construction or destruction (12.6.2). When a virtual function is called directly or indirectly from a constructor (including the *mem-initializer* or *brace-or-equal-initializer* for a non-static data member) or from a destructor, and the object to which the

call applies is the object under construction or destruction, the function called is the one defined in the constructor or destructor's own class or in one of its bases, but not a function overriding it in a class derived from the constructor or destructor's class, or overriding it in one of the other base classes of the most derived object (1.8). If the virtual function call uses an explicit class member access (5.2.5) and the object-expression refers to the object under construction or destruction but its type is neither the constructor or destructor's own class or one of its bases, the result of the call is undefined. [*Example:*

```
struct V {
  virtual void f();
  virtual void g();
};

struct A : virtual V {
  virtual void f();
};

struct B : virtual V {
  virtual void g();
  B(V*, A*);
};

struct D : A, B {
  virtual void f();
  virtual void g();
  D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
  f();               // calls V::f, not A::f
  g();               // calls B::g, not D::g
  v->g();            // v is base of B, the call is well-defined, calls B::g
  a->f();            // undefined behavior, a's type not a base of B
}
```

— *end example*]

5 The `typeid` operator (5.2.8) can be used during construction or destruction (12.6.2). When `typeid` is used in a constructor (including the *mem-initializer* or *brace-or-equal-initializer* for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of `typeid` refers to the object under construction or destruction, `typeid` yields the `std::type_info` object representing the constructor or destructor's class. If the operand of `typeid` refers to the object under construction or destruction and the static type of the operand is neither the constructor or destructor's class nor one of its bases, the result of `typeid` is undefined.

6 `Dynamic_cast`s (5.2.7) can be used during construction or destruction (12.6.2). When a `dynamic_cast` is used in a constructor (including the *mem-initializer* or *brace-or-equal-initializer* for a non-static data member) or in a destructor, or used in a function called (directly or indirectly) from a constructor or destructor, if the operand of the `dynamic_cast` refers to the object under construction or destruction, this object is considered to be a most derived object that has the type of the constructor or destructor's class. If the operand of the `dynamic_cast` refers to the object under construction or destruction and the static type of the operand is not a pointer to or object of the constructor or destructor's own class or one of its bases, the `dynamic_cast` results in undefined behavior.

[*Example:*

```
struct V {
```

```
  virtual void f();
};

struct A : virtual V { };

struct B : virtual V {
  B(V*, A*);
};

struct D : A, B {
  D() : B((A*)this, this) { }
};

B::B(V* v, A* a) {
  typeid(*this);                // type_info for B
  typeid(*v);                   // well-defined: *v has type V, a base of B
                                // yields type_info for B
  typeid(*a);                   // undefined behavior: type A not a base of B
  dynamic_cast<B*>(v);          // well-defined: v of type V*, V base of B
                                // results in B*
  dynamic_cast<B*>(a);          // undefined behavior,
                                // a has type A*, A not a base of B
}
```

— *end example*]

12.8 Copying class objects [class.copy]

1 A class object can be copied in two ways, by initialization (12.1, 8.5), including for function argument passing (5.2.2) and for function value return (6.6.3), and by assignment (5.17). Conceptually, these two operations are implemented by a copy constructor (12.1) and copy assignment operator (13.5.3).

2 A non-template constructor for class `X` is a *copy* constructor if its first parameter is of type `X&`, `const X&`, `volatile X&` or `const volatile X&`, and either there are no other parameters or else all other parameters have default arguments (8.3.6).[110] [*Example:* `X::X(const X&)` and `X::X(X&,int=1)` are copy constructors.

```
struct X {
  X(int);
  X(const X&, int = 1);
};
X a(1);             // calls X(int);
X b(a, 0);          // calls X(const X&, int);
X c = b;            // calls X(const X&, int);
```

— *end example*] [*Note:* all forms of copy constructor may be declared for a class. [*Example:*

```
struct X {
  X(const X&);
  X(X&);            // OK
};
```

110) Because a template constructor or a constructor whose first parameter is an rvalue reference is never a copy constructor, the presence of such a constructor does not suppress the implicit declaration of a copy constructor. Such constructors participate in overload resolution with other constructors, including copy constructors, and, if selected, will be used to copy an object.

— *end example*] — *end note*] [*Note:* if a class `X` only has a copy constructor with a parameter of type `X&`, an initializer of type `const X` or `volatile X` cannot initialize an object of type (possibly cv-qualified) `X`. [*Example:*

```
struct X {
  X();                  // default constructor
  X(X&);                // copy constructor with a nonconst parameter
};
const X cx;
X x = cx;               // error: X::X(X&) cannot copy cx into x
```

— *end example*] — *end note*]

3 A declaration of a constructor for a class `X` is ill-formed if its first parameter is of type (optionally cv-qualified) `X` and either there are no other parameters or else all other parameters have default arguments. A member function template is never instantiated to perform the copy of a class object to an object of its class type. [*Example:*

```
struct S {
  template<typename T> S(T);
};

S f();

void g() {
  S a( f() );           // does not instantiate member template
}
```

— *end example*]

4 If the class definition does not explicitly declare a copy constructor, one is declared *implicitly*. Thus, for the class definition

```
struct X {
  X(const X&, int);
};
```

a copy constructor is implicitly-declared. If the user-declared constructor is later defined as

```
X::X(const X& x, int i =0) { /* ... */ }
```

then any use of `X`'s copy constructor is ill-formed because of the ambiguity; no diagnostic is required.

5 The implicitly-declared copy constructor for a class `X` will have the form

```
X::X(const X&)
```

if

— each direct or virtual base class `B` of `X` has a copy constructor whose first parameter is of type `const B&` or `const volatile B&`, and

— for all the non-static data members of `X` that are of a class type `M` (or array thereof), each such class type has a copy constructor whose first parameter is of type `const M&` or `const volatile M&`.[111]

Otherwise, the implicitly-declared copy constructor will have the form

111) This implies that the reference parameter of the implicitly-declared copy constructor cannot bind to a `volatile` lvalue; see C.1.8.

`X::X(X&)`

An implicitly-declared copy constructor is an `inline public` member of its class. An implicitly-declared copy constructor for a class `X` is defined as deleted if `X` has:

— a variant member with a non-trivial copy constructor and `X` is a union-like class,

— a non-static data member of class type `M` (or array thereof) that cannot be copied because overload resolution (13.3), as applied to `M`'s copy constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy constructor, or

— a direct or virtual base class `B` that cannot be copied because overload resolution (13.3), as applied to `B`'s copy constructor, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy constructor.

6 A copy constructor for class `X` is *trivial* if it is not user-provided (8.4) and if

— class `X` has no virtual functions (10.3) and no virtual base classes (10.1), and

— the constructor selected to copy each direct base class subobject is trivial, and

— for each non-static data memberof `X` that is of class type (or array thereof), the constructor selected to copy that member is trivial;

otherwise the copy constructor is *non-trivial*.

7 A non-user-provided copy constructor is *implicitly defined* if it is used to initialize an object of its class type from a copy of an object of its class type or of a class type derived from its class type[112]. [*Note:* the copy constructor is implicitly defined even if the implementation elided its use (12.2). — *end note*] A program is ill-formed if the implicitly-defined copy constructor is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

Before the non-user-provided copy constructor for a class is implicitly defined, all non-user-provided copy constructors for its direct and virtual base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared copy constructor has an *exception-specification* (15.4). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

8 The implicitly-defined or explicitly-defaulted copy constructor for class `X` performs a memberwise copy of its subobjects. [*Note: brace-or-equal-initializers* of non-static data members are ignored. See also the example in 12.6.2. — *end note*] The order of copying is the same as the order of initialization of bases and members in a user-defined constructor (see 12.6.2). Each subobject is copied in the manner appropriate to its type:

— if the subobject is of class type, the copy constructor for the class is used;

— if the subobject is an array, each element is copied, in the manner appropriate to the element type;

— if the subobject is of scalar type, the built-in assignment operator is used.

Virtual base class subobjects shall be copied only once by the implicitly-defined copy constructor (see 12.6.2).

9 A user-declared *copy* assignment operator `X::operator=` is a non-static non-template member function of class `X` with exactly one parameter of type `X`, `X&`, `const X&`, `volatile X&` or `const volatile X&`.[113] [*Note:* an overloaded assignment operator must be declared to have only one parameter; see 13.5.3. — *end note*] [*Note:* more than one form of copy assignment operator may be declared for a class. — *end note*] [*Note:*

112) See 8.5 for more details on direct and copy initialization.

113) Because a template assignment operator or an assignment operator taking an rvalue reference parameter is never a copy assignment operator, the presence of such an assignment operator does not suppress the implicit declaration of a copy assignment operator. Such assignment operators participate in overload resolution with other assignment operators, including copy assignment operators, and, if selected, will be used to assign an object.

if a class X only has a copy assignment operator with a parameter of type X&, an expression of type const X cannot be assigned to an object of type X. [*Example:*

```
struct X {
  X();
  X& operator=(X&);
};
const X cx;
X x;
void f() {
  x = cx;             // error: X::operator=(X&) cannot assign cx into x
}
```

— *end example*] — *end note*]

10 If the class definition does not explicitly declare a copy assignment operator, one is declared *implicitly*. The implicitly-declared copy assignment operator for a class X will have the form

```
X& X::operator=(const X&)
```

if

— each direct base class B of X has a copy assignment operator whose parameter is of type const B&, const volatile B& or B, and

— for all the non-static data members of X that are of a class type M (or array thereof), each such class type has a copy assignment operator whose parameter is of type const M&, const volatile M& or M.[114]

Otherwise, the implicitly-declared copy assignment operator will have the form

```
X& X::operator=(X&)
```

The implicitly-declared copy assignment operator for class X has the return type X&; it returns the object for which the assignment operator is invoked, that is, the object assigned to. An implicitly-declared copy assignment operator is an inline public member of its class. An implicitly-declared copy assignment operator for class X is defined as deleted if X has:

— a variant member with a non-trivial copy assignment operator and X is a union-like class,

— a non-static data member of const non-class type (or array thereof), or

— a non-static data member of reference type, or

— a non-static data member of class type M (or array thereof) that cannot be copied because overload resolution (13.3), as applied to M's copy assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy assignment operator, or

— a direct or virtual base class B that cannot be copied because overload resolution (13.3), as applied to B's copy assignment operator, results in an ambiguity or a function that is deleted or inaccessible from the implicitly-declared copy assignment operator.

Because a copy assignment operator is implicitly declared for a class if not declared by the user, a base class copy assignment operator is always hidden by the copy assignment operator of a derived class (13.5.3). A *using-declaration* (7.3.3) that brings in from a base class an assignment operator with a parameter type that could be that of a copy-assignment operator for the derived class is not considered an explicit declaration

114) This implies that the reference parameter of the implicitly-declared copy assignment operator cannot bind to a volatile lvalue; see C.1.8.

of a copy-assignment operator and does not suppress the implicit declaration of the derived class copy-assignment operator; the operator introduced by the *using-declaration* is hidden by the implicitly-declared copy-assignment operator in the derived class.

11 A copy assignment operator for class `X` is *trivial* if it is not user-provided and if

- — class `X` has no virtual functions (10.3) and no virtual base classes (10.1), and
- — the assignment operator selected to copy each direct base class subobject is trivial, and
- — for each non-static data memberof `X` that is of class type (or array thereof), the assignment operator selected to copy that member is trivial;

otherwise the copy assignment operator is *non-trivial*.

12 A non-user-provided copy assignment operator is *implicitly defined* when an object of its class type is assigned a value of its class type or a value of a class type derived from its class type. A program is ill-formed if the implicitly-defined copy assignment operator is explicitly defaulted, but the corresponding implicit declaration would have been deleted.

Before the non-user-provided copy assignment operator for a class is implicitly defined, all non-user-provided copy assignment operators for its direct base classes and its non-static data members shall have been implicitly defined. [*Note:* an implicitly-declared copy assignment operator has an *exception-specification* (15.4). An explicitly-defaulted definition has no implicit *exception-specification*. — *end note*]

13 The implicitly-defined or explicitly-defaulted copy assignment operator for class `X` performs memberwise assignment of its subobjects. The direct base classes of `X` are assigned first, in the order of their declaration in the *base-specifier-list*, and then the immediate non-static data members of `X` are assigned, in the order in which they were declared in the class definition. Each subobject is assigned in the manner appropriate to its type:

- — if the subobject is of class type, the copy assignment operator for the class is used (as if by explicit qualification; that is, ignoring any possible virtual overriding functions in more derived classes);
- — if the subobject is an array, each element is assigned, in the manner appropriate to the element type;
- — if the subobject is of scalar type, the built-in assignment operator is used.

It is unspecified whether subobjects representing virtual base classes are assigned more than once by the implicitly-defined or explicitly-defaulted copy assignment operator. [*Example:*

```
struct V { };
struct A : virtual V { };
struct B : virtual V { };
struct C : B, A { };
```

It is unspecified whether the virtual base class subobject `V` is assigned twice by the implicitly-defined copy assignment operator for `C`. — *end example*]

14 A program is ill-formed if the copy constructor or the copy assignment operator for an object is implicitly used and the special member function is not accessible (Clause 11). [*Note:* Copying one object into another using the copy constructor or the copy assignment operator does not change the layout or size of either object. — *end note*]

15 When certain criteria are met, an implementation is allowed to omit the copy construction of a class object, even if the copy constructor and/or destructor for the object have side effects. In such cases, the implementation treats the source and target of the omitted copy operation as simply two different ways of referring to the same object, and the destruction of that object occurs at the later of the times when the two objects

would have been destroyed without the optimization.[115] This elision of copy operations is permitted in the following circumstances (which may be combined to eliminate multiple copies):

— in a `return` statement in a function with a class return type, when the expression is the name of a non-volatile automatic object with the same cv-unqualified type as the function return type, the copy operation can be omitted by constructing the automatic object directly into the function's return value

— in a *throw-expression*, when the operand is the name of a non-volatile automatic object, the copy operation from the operand to the exception object (15.1) can be omitted by constructing the automatic object directly into the exception object

— when a temporary class object that has not been bound to a reference (12.2) would be copied to a class object with the same cv-unqualified type, the copy operation can be omitted by constructing the temporary object directly into the target of the omitted copy

— when the *exception-declaration* of an exception handler (Clause 15) declares an object of the same type (except for cv-qualification) as the exception object (15.1), the copy operation can be omitted by treating the *exception-declaration* as an alias for the exception object if the meaning of the program will be unchanged except for the execution of constructors and destructors for the object declared by the *exception-declaration*.

[*Example:*

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(const Thing&);
};

Thing f() {
  Thing t;
  return t;
}

Thing t2 = f();
```

Here the criteria for elision can be combined to eliminate two calls to the copy constructor of class `Thing`: the copying of the local automatic object `t` into the temporary object for the return value of function `f()` and the copying of that temporary object into object `t2`. Effectively, the construction of the local object `t` can be viewed as directly initializing the global object `t2`, and that object's destruction will occur at program exit. — *end example*]

16 When the criteria for elision of a copy operation are met and the object to be copied is designated by an lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue. If overload resolution fails, or if the type of the first parameter of the selected constructor is not an rvalue reference to the object's type (possibly cv-qualified), overload resolution is performed again, considering the object as an lvalue. [*Note:* This two-stage overload resolution must be performed regardless of whether copy elision will occur. It determines the constructor to be called if elision is not performed, and the selected constructor must be accessible even if the call is elided. — *end note*]

[*Example:*

115) Because only one object is destroyed instead of two, and one copy constructor is not executed, there is still one object destroyed for each one constructed.

```
class Thing {
public:
  Thing();
  ~Thing();
  Thing(Thing&&);
private:
  Thing(const Thing&);
};

Thing f(bool b) {
  Thing t;
  if (b)
    throw t;                    // OK: Thing(Thing&&) used (or elided) to throw t
  return t;                     // OK: Thing(Thing&&) used (or elided) to return t
}

Thing t2 = f(false);            // OK: Thing(Thing&&) used (or elided) to construct of t2
```

— *end example*]

12.9 Inheriting Constructors [class.inhctor]

1 A *using-declaration* (7.3.3) that names a constructor implicitly declares a set of *inheriting constructors*. The *candidate set of inherited constructors* from the class `X` named in the *using-declaration* consists of actual constructors and notional constructors that result from the transformation of defaulted parameters as follows:

- all non-template constructors of `X`, and
- for each non-template constructor of `X` that has at least one parameter with a default argument, the set of constructors that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list, and
- all constructor templates of `X`, and
- for each constructor template of `X` that has at least one parameter with a default argument, the set of constructor templates that results from omitting any ellipsis parameter specification and successively omitting parameters with a default argument from the end of the parameter-type-list.

2 The *constructor characteristics* of a constructor or constructor template are

- the template parameter list (14.1), if any,
- the template requirements (14.10.1), if any,
- the *parameter-type-list* (8.3.5),
- the *exception-specification* (15.4),
- absence or presence of `explicit` (12.3.1), and
- absence or presence of `constexpr` (7.1.5).

3 For each non-template constructor in the candidate set of inherited constructors other than a constructor having no parameters or a copy constructor having a single parameter, a constructor is implicitly declared with the same constructor characteristics unless there is a user-declared constructor with the same signature in the class where the *using-declaration* appears. Similarly, for each constructor template in the candidate

set of inherited constructors, a constructor template is implicitly declared with the same constructor characteristics unless there is an equivalent user-declared constructor template (14.5.6.1) in the class where the using-declaration appears. [*Note:* Default arguments are not inherited. — *end note*]

4 A constructor so declared has the same access as the corresponding constructor in `X`. It is deleted if the corresponding constructor in `X` is deleted (8.4).

5 [*Note:* Default and copy constructors may be implicitly declared as specified in 12.1 and 12.8. — *end note*]

6 [*Example:*

```
struct B1 {
  B1(int);
};

struct B2 {
  B2(int = 13, int = 42);
};

struct D1 : B1 {
  using B1::B1;
};

struct D2 : B2 {
  using B2::B2;
};
```

The candidate set of inherited constructors in `D1` for `B1` is

— `B1(const B1&)`

— `B1(int)`

The set of constructors present in `D1` is

— `D1()`, implicitly-declared default constructor, ill-formed if used

— `D1(const D1&)`, implicitly-declared copy constructor, not inherited

— `D1(int)`, implicitly-declared inheriting constructor

The candidate set of inherited constructors in `D2` for `B2` is

— `B2(const B2&)`

— `B2(int = 13, int = 42)`

— `B2(int = 13)`

— `B2()`

The set of constructors present in `D2` is

— `D2()`, implicitly-declared default constructor, not inherited

— `D2(const D2&)`, implicitly-declared copy constructor, not inherited

— `D2(int, int)`, implicitly-declared inheriting constructor

— `D2(int)`, implicitly-declared inheriting constructor

— *end example*]

7 [*Note:* If two *using-declaration*s declare inheriting constructors with the same signatures, the program is ill-formed (9.2, 13.1), because an implicitly-declared constructor introduced by the first *using-declaration* is not a user-declared constructor and thus does not preclude another declaration of a constructor with the same signature by a subsequent *using-declaration*. [*Example:*

```
struct B1 {
  B1(int);
};

struct B2 {
  B2(int);
};

struct D1 : B1, B2 {
  using B1::B1;
  using B2::B2;
};                  // ill-formed: attempts to declare D1(int) twice

struct D2 : B1, B2 {
  using B1::B1;
  using B2::B2;
  D2(int);          // OK: user declaration supersedes both implicit declarations
};
```

— *end example*] — *end note*]

8 An inheriting constructor for a class is implicitly defined when it is used (3.2) to create an object of its class type (1.8). An implicitly-defined inheriting constructor performs the set of initializations of the class that would be performed by a user-written `inline` constructor for that class with a *mem-initializer-list* whose only *mem-initializer* has a *mem-initializer-id* that names the base class named in the *nested-name-specifier* of the *using-declaration* and an *expression-list* as specified below, and where the *compound-statement* in its function body is empty (12.6.2). If that user-written constructor would be ill-formed, the program is ill-formed. Each *expression* in the *expression-list* is of the form `static_cast<T&&>(p)`, where `p` is the name of the corresponding constructor parameter and `T` is the declared type of `p`.

9 [*Example:*

```
struct B1 {
  B1(int) { }
};

struct B2 {
  B2(double) { }
};

struct D1 : B1 {
  using B1::B1;     // implicitly declares D1(int)
  int x;
};

void test() {
  D1 d(6);          // OK: d.x is not initialized
  D1 e;             // error: D1 has no default constructor
}
```

```
struct D2 : B2 {
  using B2::B2;      // OK: implicitly declares D2(double)
  B1 b;
};

D2 f(1.0);           // error: B1 has no default constructor

template< class T >
struct D : T {
  using T::T;        // declares all constructors from class T
  ~D() { std::clog << "Destroying wrapper" << std::endl; }
};
```

Class template `D` wraps any class and forwards all of its contructors, while writing a message to the standard log whenever an object of class `D` is destroyed. — *end example*]

13 Overloading [over]

1 When two or more different declarations are specified for a single name in the same scope, that name is said to be *overloaded*. By extension, two declarations in the same scope that declare the same name but with different types are called *overloaded declarations*. Only function declarations can be overloaded; object and type declarations cannot be overloaded.

2 When an overloaded function name is used in a call, which overloaded function declaration is being referenced is determined by comparing the types of the arguments at the point of use with the types of the parameters in the overloaded declarations that are visible at the point of use. This function selection process is called *overload resolution* and is defined in 13.3. [*Example:*

```
double abs(double);
int abs(int);

abs(1);             // calls abs(int);
abs(1.0);           // calls abs(double);
```

— *end example*]

13.1 Overloadable declarations [over.load]

1 Not all function declarations can be overloaded. Those that cannot be overloaded are specified here. A program is ill-formed if it contains two such non-overloadable declarations in the same scope. [*Note:* this restriction applies to explicit declarations in a scope, and between such declarations and declarations made through a *using-declaration* (7.3.3). It does not apply to sets of functions fabricated as a result of name lookup (e.g., because of *using-directives*) or overload resolution (e.g., for operator functions). — *end note*]

2 Certain function declarations cannot be overloaded:

— Function declarations that differ only in the return type cannot be overloaded.

— Member function declarations with the same name, the same *parameter-type-list*, and the same template requirements (if any) cannot be overloaded if any of them is a `static` member function declaration (9.4). Likewise, member function template declarations with the same name, the same *parameter-type-list*, the same template parameter lists, and the same template requirements (if any) cannot be overloaded if any of them is a `static` member function template declaration. The types of the implicit object parameters constructed for the member functions for the purpose of overload resolution (13.3.1) are not considered when comparing parameter-type-lists for enforcement of this rule. In contrast, if there is no `static` member function declaration among a set of member function declarations with the same name and the same parameter-type-list, then these member function declarations can be overloaded if they differ in the type of their implicit object parameter. [*Example:* the following illustrates this distinction:

```
class X {
  static void f();
  void f();                     // ill-formed
  void f() const;               // ill-formed
  void f() const volatile;      // ill-formed
  void g();
  void g() const;               // OK: no static g
```

```
  void g() const volatile;      // OK: no static g
};
```

— *end example*]

— Member function declarations with the same name and the same *parameter-type-list* as well as member function template declarations with the same name, the same *parameter-type-list*, the same template parameter lists, and the same template requirements cannot be overloaded if any of them, but not all, have a *ref-qualifier* (8.3.5). [*Example:*

```
class Y {
  void h() &;
  void h() const &;             // OK
  void h() &&;                  // OK, all declarations have a ref-qualifier
  void i() &;
  void i() const;               // ill-formed, prior declaration of i
                                // has a ref-qualifier
};
```

— *end example*]

3 [*Note:* as specified in 8.3.5, function declarations that have equivalent parameter declarations declare the same function and therefore cannot be overloaded:

— Parameter declarations that differ only in the use of equivalent typedef "types" are equivalent. A `typedef` is not a separate type, but only a synonym for another type (7.1.3). [*Example:*

```
typedef int Int;

void f(int i);
void f(Int i);                  // OK: redeclaration of f(int)
void f(int i) { /* ... */ }
void f(Int i) { /* ... */ }     // error: redefinition of f(int)
```

— *end example*]

Enumerations, on the other hand, are distinct types and can be used to distinguish overloaded function declarations. [*Example:*

```
enum E { a };

void f(int i) { /* ... */ }
void f(E i)   { /* ... */ }
```

— *end example*]

— Parameter declarations that differ only in a pointer `*` versus an array `[]` are equivalent. That is, the array declaration is adjusted to become a pointer declaration (8.3.5). Only the second and subsequent array dimensions are significant in parameter types (8.3.4). [*Example:*

```
int f(char*);
int f(char[]);                  // same as f(char*);
int f(char[7]);                 // same as f(char*);
int f(char[9]);                 // same as f(char*);

int g(char(*)[10]);
int g(char[5][10]);             // same as g(char(*)[10]);
int g(char[7][10]);             // same as g(char(*)[10]);
```

```
int g(char(*)[20]);             // different from g(char(*)[10]);
```

— *end example*]

— Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent. That is, the function type is adjusted to become a pointer to function type (8.3.5). [*Example:*

```
void h(int());
void h(int (*)());              // redeclaration of h(int())
void h(int x()) { }             // definition of h(int())
void h(int (*x)()) { }          // ill-formed: redefinition of h(int())
```

— *end example*]

— Parameter declarations that differ only in the presence or absence of `const` and/or `volatile` are equivalent. That is, the `const` and `volatile` type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. [*Example:*

```
typedef const int cInt;

int f (int);
int f (const int);              // redeclaration of f(int)
int f (int) { ... }             // definition of f(int)
int f (cInt) { ... }            // error: redefinition of f(int)
```

— *end example*]

Only the `const` and `volatile` type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; `const` and `volatile` type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations.[116] In particular, for any type `T`, "pointer to `T`," "pointer to `const T`," and "pointer to `volatile T`" are considered distinct parameter types, as are "reference to `T`," "reference to `const T`," and "reference to `volatile T`."

— Two parameter declarations that differ only in their default arguments are equivalent. [*Example:* consider the following:

```
void f (int i, int j);
void f (int i, int j = 99);     // OK: redeclaration of f(int, int)
void f (int i = 88, int j);     // OK: redeclaration of f(int, int)
void f ();                      // OK: overloaded declaration of f

void prog () {
    f (1, 2);                   // OK: call f(int, int)
    f (1);                      // OK: call f(int, int)
    f ();                       // Error: f(int, int) or f()?
}
```

— *end example*] — *end note*]

116) When a parameter type includes a function type, such as in the case of a parameter type that is a pointer to function, the `const` and `volatile` type-specifiers at the outermost level of the parameter type specifications for the inner function type are also ignored.

13.2 Declaration matching [over.dcl]

1 Two function declarations of the same name refer to the same function if they are in the same scope and have equivalent parameter declarations (13.1). A function member of a derived class is *not* in the same scope as a function member of the same name in a base class. [*Example:*

```
struct B {
  int f(int);
};

struct D : B {
  int f(char*);
};
```

Here `D::f(char*)` hides `B::f(int)` rather than overloading it.

```
void h(D* pd) {
  pd->f(1);                       // error:
                                  // D::f(char*) hides B::f(int)
  pd->B::f(1);                    // OK
  pd->f("Ben");                   // OK, calls D::f
}
```

— *end example*]

2 A locally declared function is not in the same scope as a function in a containing scope. [*Example:*

```
void f(char*);
void g() {
  extern void f(int);
  f("asdf");                      // error: f(int) hides f(char*)
                                  // so there is no f(char*) in this scope
}

void caller () {
  extern void callee(int, int);
  {
    extern void callee(int);      // hides callee(int, int)
    callee(88, 99);               // error: only callee(int) in scope
  }
}
```

— *end example*]

3 Different versions of an overloaded member function can be given different access rules. [*Example:*

```
class buffer {
private:
    char* p;
    int size;
protected:
    buffer(int s, char* store) { size = s; p = store; }
public:
    buffer(int s) { p = new char[size = s]; }
};
```

— *end example*]

13.3 Overload resolution [over.match]

1 Overload resolution is a mechanism for selecting the best function to call given a list of expressions that are to be the arguments of the call and a set of *candidate functions* that can be called based on the context of the call. The selection criteria for the best function are the number of arguments, how well the arguments match the parameter-type-list of the candidate function, how well (for non-static member functions) the object matches the implied object parameter, and certain other properties of the candidate function. [*Note:* the function selected by overload resolution is not guaranteed to be appropriate for the context. Other restrictions, such as the accessibility of the function, can make its use in the calling context ill-formed. — *end note*]

2 Overload resolution selects the function to call in seven distinct contexts within the language:

— invocation of a function named in the function call syntax (13.3.1.1.1);

— invocation of a function call operator, a pointer-to-function conversion function, a reference-to-pointer-to-function conversion function, or a reference-to-function conversion function on a class object named in the function call syntax (13.3.1.1.2);

— invocation of the operator referenced in an expression (13.3.1.2);

— invocation of a constructor for direct-initialization (8.5) of a class object (13.3.1.3);

— invocation of a user-defined conversion for copy-initialization (8.5) of a class object (13.3.1.4);

— invocation of a conversion function for initialization of an object of a nonclass type from an expression of class type (13.3.1.5); and

— invocation of a conversion function for conversion to an lvalue to which a reference (8.5.3) will be directly bound (13.3.1.6).

Each of these contexts defines the set of candidate functions and the list of arguments in its own unique way. But, once the candidate functions and argument lists have been identified, the selection of the best function is the same in all cases:

— First, a subset of the candidate functions (those that have the proper number of arguments and meet certain other conditions) is selected to form a set of viable functions (13.3.2).

— Then the best viable function is selected based on the implicit conversion sequences (13.3.3.1) needed to match each argument to the corresponding parameter of each viable function.

3 If a best viable function exists and is unique, overload resolution succeeds and produces it as the result. Otherwise overload resolution fails and the invocation is ill-formed. When overload resolution succeeds, and the best viable function is not accessible (Clause 11) in the context in which it is used, the program is ill-formed.

13.3.1 Candidate functions and argument lists [over.match.funcs]

1 The subclauses of 13.3.1 describe the set of candidate functions and the argument list submitted to overload resolution in each of the seven contexts in which overload resolution is used. [*Note:* With concepts (14.9) and constrained templates, the set of candidate functions can be determined by an associated function candidate set or a retained candidate set (14.10.3). — *end note*] The source transformations and constructions defined in these subclauses are only for the purpose of describing the overload resolution process. An implementation is not required to use such transformations and constructions.

2 The set of candidate functions can contain both member and non-member functions to be resolved against the same argument list. So that argument and parameter lists are comparable within this heterogeneous set, a member function is considered to have an extra parameter, called the *implicit object parameter*, which represents the object for which the member function has been called. For the purposes of overload resolution, both static and non-static member functions have an implicit object parameter, but constructors do not.

3 Similarly, when appropriate, the context can construct an argument list that contains an *implied object argument* to denote the object to be operated on. Since arguments and parameters are associated by position within their respective lists, the convention is that the implicit object parameter, if present, is always the first parameter and the implied object argument, if present, is always the first argument.

4 For non-static member functions, the type of the implicit object parameter is

— "lvalue reference to *cv* `X`" for functions declared without a *ref-qualifier* or with the `&` *ref-qualifier*

— "rvalue reference to *cv* `X`" for functions declared with the `&&` *ref-qualifier*

where `X` is the class of which the function is a member and *cv* is the cv-qualification on the member function declaration. [*Example:* for a `const` member function of class `X`, the extra parameter is assumed to have type "reference to `const X`". — *end example*] For conversion functions, the function is considered to be a member of the class of the implicit object argument for the purpose of defining the type of the implicit object parameter. For non-conversion functions introduced by a *using-declaration* into a derived class, the function is considered to be a member of the derived class for the purpose of defining the type of the implicit object parameter. For static member functions, the implicit object parameter is considered to match any object (since if the function is selected, the object is discarded). [*Note:* no actual type is established for the implicit object parameter of a static member function, and no attempt will be made to determine a conversion sequence for that parameter (13.3.3). — *end note*]

5 During overload resolution, the implied object argument is indistinguishable from other arguments. The implicit object parameter, however, retains its identity since conversions on the corresponding argument shall obey these additional rules:

— no temporary object can be introduced to hold the argument for the implicit object parameter; and

— no user-defined conversions can be applied to achieve a type match with it.

For non-static member functions declared without a *ref-qualifier*, an additional rule applies:

— even if the implicit object parameter is not `const`-qualified, an rvalue temporary can be bound to the parameter as long as in all other respects the temporary can be converted to the type of the implicit object parameter. [*Note:* The fact that such a temporary is an rvalue does not affect the ranking of implicit conversion sequences (13.3.3.2). — *end note*]

6 Because other than in list-initialization only one user-defined conversion is allowed in an implicit conversion sequence, special rules apply when selecting the best user-defined conversion (13.3.3, 13.3.3.1). [*Example:*

```
class T {
public:
  T();
};

class C : T {
public:
  C(int);
};
T a = 1;            // ill-formed: T(C(1)) not tried
```

— *end example*]

7 In each case where a candidate is a function template, candidate function template specializations are generated using template argument deduction (14.8.3, 14.8.2). Those candidates are then handled as candidate functions in the usual way.[117] A given name can refer to one or more function templates and also to a set of overloaded non-template functions. In such a case, the candidate functions generated from each function template are combined with the set of non-template candidate functions.

13.3.1.1 Function call syntax [over.match.call]

1 Recall from 5.2.2, that a *function call* is a *postfix-expression*, possibly nested arbitrarily deep in parentheses, followed by an optional *expression-list* enclosed in parentheses:

(... ($_{opt}$ *postfix-expression*) ...) $_{opt}$ (*expression-list* $_{opt}$)

Overload resolution is required if the *postfix-expression* is the name of a function, a function template (14.5.6), an object of class type, or a set of pointers-to-function.

2 13.3.1.1.1 describes how overload resolution is used in the first two of the above cases to determine the function to call. 13.3.1.1.2 describes how overload resolution is used in the third of the above cases to determine the function to call.

3 The fourth case arises from a *postfix-expression* of the form `&F`, where `F` names a set of overloaded functions. In the context of a function call, `&F` is treated the same as the name `F` by itself. Thus, `(&F)(` *expression-list* $_{opt}$ `)` is simply `(F)(` *expression-list* $_{opt}$ `)`, which is discussed in 13.3.1.1.1. If the function selected by overload resolution according to 13.3.1.1.1 is a non-static member function, the program is ill-formed.[118] (The resolution of `&F` in other contexts is described in 13.4.)

13.3.1.1.1 Call to named function [over.call.func]

1 Of interest in 13.3.1.1.1 are only those function calls in which the *postfix-expression* ultimately contains a name that denotes one or more functions that might be called. Such a *postfix-expression*, perhaps nested arbitrarily deep in parentheses, has one of the following forms:

postfix-expression:
postfix-expression `.` *id-expression*
postfix-expression `->` *id-expression*
primary-expression

These represent two syntactic subcategories of function calls: qualified function calls and unqualified function calls.

2 In qualified function calls, the name to be resolved is an *id-expression* and is preceded by an `->` or `.` operator. Since the construct `A->B` is generally equivalent to `(*A).B`, the rest of Clause 13 assumes, without loss of generality, that all member function calls have been normalized to the form that uses an object and the `.` operator. Furthermore, Clause 13 assumes that the *postfix-expression* that is the left operand of the `.` operator has type "*cv* `T`" where `T` denotes a class[119]. Under this assumption, the *id-expression* in the call is looked up as a member function of `T` following the rules for looking up names in classes (10.2). The function declarations found by that lookup constitute the set of candidate functions. The argument list is the *expression-list* in the call augmented by the addition of the left operand of the `.` operator in the normalized member function call as the implied object argument (13.3.1).

117) The process of argument deduction fully determines the parameter types of the function template specializations, i.e., the parameters of function template specializations contain no template parameter types. Therefore the function template specializations can be treated as normal (non-template) functions for the remainder of overload resolution.
118) When `F` is a non-static member function, a reference of the form `&A::f` is a pointer-to-member, which cannot be used with the function-call syntax, and a reference of the form `&F` is an invalid use of the "`&`" operator on a non-static member function.
119) Note that cv-qualifiers on the type of objects are significant in overload resolution for both lvalue and class rvalue objects.

3 In unqualified function calls, the name is not qualified by an `->` or `.` operator and has the more general form of a *primary-expression*. The name is looked up in the context of the function call following the normal rules for name lookup in function calls (3.4). The function declarations found by that lookup constitute the set of candidate functions. Because of the rules for name lookup, the set of candidate functions consists (1) entirely of non-member functions or (2) entirely of member functions of some class `T`. In case (1), the argument list is the same as the *expression-list* in the call. In case (2), the argument list is the *expression-list* in the call augmented by the addition of an implied object argument as in a qualified function call. If the keyword `this` (9.3.2) is in scope and refers to class `T`, or a derived class of `T`, then the implied object argument is `(*this)`. If the keyword `this` is not in scope or refers to another class, then a contrived object of type `T` becomes the implied object argument[120]. If the argument list is augmented by a contrived object and overload resolution selects one of the non-static member functions of `T`, the call is ill-formed.

13.3.1.1.2 Call to object of class type [over.call.object]

1 If the *primary-expression* `E` in the function call syntax evaluates to a class object of type "*cv* `T`", then the set of candidate functions includes at least the function call operators of `T`. The function call operators of `T` are obtained by ordinary lookup of the name `operator()` in the context of `(E).operator()`.

2 In addition, for each non-explicit conversion function declared in `T` of the form

`operator` *conversion-type-id* `()` *attribute-specifier*$_{opt}$ *cv-qualifier* `;`

where *cv-qualifier* is the same cv-qualification as, or a greater cv-qualification than, *cv*, and where *conversion-type-id* denotes the type "pointer to function of `(P1,...,Pn)` returning `R`", or the type "reference to pointer to function of `(P1,...,Pn)` returning `R`", or the type "reference to function of `(P1,...,Pn)` returning `R`", a *surrogate call function* with the unique name *call-function* and having the form

`R` *call-function* `(` *conversion-type-id* `F, P1 a1, ... ,Pn an) { return F (a1,... ,an); }`

is also considered as a candidate function. Similarly, surrogate call functions are added to the set of candidate functions for each non-explicit conversion function declared in a base class of `T` provided the function is not hidden within `T` by another intervening declaration[121].

3 If such a surrogate call function is selected by overload resolution, the corresponding conversion function will be called to convert `E` to the appropriate function pointer or reference, and the function will then be invoked with the arguments of the call. If the conversion function cannot be called (e.g., because of an ambiguity), the program is ill-formed.

4 The argument list submitted to overload resolution consists of the argument expressions present in the function call syntax preceded by the implied object argument `(E)`. [*Note:* when comparing the call against the function call operators, the implied object argument is compared against the implicit object parameter of the function call operator. When comparing the call against a surrogate call function, the implied object argument is compared against the first parameter of the surrogate call function. The conversion function from which the surrogate call function was derived will be used in the conversion sequence for that parameter since it converts the implied object argument to the appropriate function pointer or reference required by that first parameter. — *end note*] [*Example:*

```
int f1(int);
int f2(float);
typedef int (*fp1)(int);
```

120) An implied object argument must be contrived to correspond to the implicit object parameter attributed to member functions during overload resolution. It is not used in the call to the selected function. Since the member functions all have the same implicit object parameter, the contrived object will not be the cause to select or reject a function.

121) Note that this construction can yield candidate call functions that cannot be differentiated one from the other by overload resolution because they have identical declarations or differ only in their return type. The call will be ambiguous if overload resolution cannot select a match to the call that is uniquely better than such undifferentiable functions.

```
typedef int (*fp2)(float);
struct A {
  operator fp1() { return f1; }
  operator fp2() { return f2; }
} a;
int i = a(1);        // calls f1 via pointer returned from
                     // conversion function
```

— *end example*]

13.3.1.2 Operators in expressions [over.match.oper]

1 If no operand of an operator in an expression has a type that is a class or an enumeration, the operator is assumed to be a built-in operator and interpreted according to Clause 5. [*Note:* because `.`, `.*`, and `::` cannot be overloaded, these operators are always built-in operators interpreted according to Clause 5. `?:` cannot be overloaded, but the rules in this subclause are used to determine the conversions to be applied to the second and third operands when they have class or enumeration type (5.16). — *end note*] [*Example:*

```
struct String {
  String (const String&);
  String (char*);
  operator char* ();
};
String operator + (const String&, const String&);

void f(void) {
 char* p= "one" + "two";        // ill-formed because neither
                                // operand has user-defined type
 int I = 1 + 1;                 // Always evaluates to 2 even if
                                // user-defined types exist which
                                // would perform the operation.
}
```

— *end example*]

2 If either operand has a type that is a class or an enumeration, a user-defined operator function might be declared that implements this operator or a user-defined conversion can be necessary to convert the operand to a type that is appropriate for a built-in operator. In this case, overload resolution is used to determine which operator function or built-in operator is to be invoked to implement the operator. Therefore, the operator notation is first transformed to the equivalent function-call notation as summarized in Table 10 (where `@` denotes one of the operators covered in the specified subclause).

Table 10 — Relationship between operator and function call notation

Subclause	Expression	As member function	As non-member function
13.5.1	@a	(a).operator@ ( )	operator@ (a)
13.5.2	a@b	(a).operator@ (b)	operator@ (a, b)
13.5.3	a=b	(a).operator= (b)	
13.5.5	a[b]	(a).operator[](b)	
13.5.6	a->	(a).operator-> ( )	
13.5.7	a@	(a).operator@ (0)	operator@ (a, 0)

3 For a unary operator `@` with an operand of a type whose cv-unqualified version is `T1`, and for a binary operator `@` with a left operand of a type whose cv-unqualified version is `T1` and a right operand of a type

whose cv-unqualified version is `T2`, three sets of candidate functions, designated *member candidates*, *non-member candidates* and *built-in candidates*, are constructed as follows:

— If `T1` is a complete class type, the set of member candidates is the result of the qualified lookup of `T1::operator@` (13.3.1.1.1); otherwise, the set of member candidates is empty.

— The set of non-member candidates is the result of the unqualified lookup of `operator@` in the context of the expression according to the usual rules for name lookup in unqualified function calls (3.4.2) except that all member functions are ignored. However, if no operand has a class type, only those non-member functions in the lookup set that have a first parameter of type `T1` or "reference to (possibly cv-qualified) `T1`", when `T1` is an enumeration type, or (if there is a right operand) a second parameter of type `T2` or "reference to (possibly cv-qualified) `T2`", when `T2` is an enumeration type, are candidate functions.

— For the operator `,`, the unary operator `&`, or the operator `->`, the built-in candidates set is empty. For all other operators, the built-in candidates include all of the candidate operator functions defined in 13.6 that, compared to the given operator,

  — have the same operator name, and

  — accept the same number of operands, and

  — accept operand types to which the given operand or operands can be converted according to 13.3.3.1, and

  — do not have the same parameter-type-list as any non-template non-member candidate.

4 For the built-in assignment operators, conversions of the left operand are restricted as follows:

— no temporaries are introduced to hold the left operand, and

— no user-defined conversions are applied to the left operand to achieve a type match with the left-most parameter of a built-in candidate.

5 For all other operators, no such restrictions apply.

6 The set of candidate functions for overload resolution is the union of the member candidates, the non-member candidates, and the built-in candidates. The argument list contains all of the operands of the operator. The best function from the set of candidate functions is selected according to 13.3.2 and 13.3.3.[122] [*Example:*

```
struct A {
  operator int();
};
A operator+(const A&, const A&);
void m() {
  A a, b;
  a + b;            // operator+(a,b) chosen over int(a) + int(b)
}
```

— *end example*]

7 If a built-in candidate is selected by overload resolution, the operands are converted to the types of the corresponding parameters of the selected operation function. Then the operator is treated as the corresponding built-in operator and interpreted according to Clause 5.

122) If the set of candidate functions is empty, overload resolution is unsuccessful.

8 The second operand of operator `->` is ignored in selecting an `operator->` function, and is not an argument when the `operator->` function is called. When `operator->` returns, the operator `->` is applied to the value returned, with the original second operand.[123]

9 If the operator is the operator `,`, the unary operator `&`, or the operator `->`, and there are no viable functions, then the operator is assumed to be the built-in operator and interpreted according to Clause 5.

10 [*Note:* the lookup rules for operators in expressions are different than the lookup rules for operator function names in a function call, as shown in the following example:

```
struct A { };
void operator + (A, A);

struct B {
  void operator + (B);
  void f ();
};

A a;

void B::f() {
  operator+ (a,a);              // error: global operator hidden by member
  a + a;                        // OK: calls global operator+
}
```

— *end note*]

13.3.1.3 Initialization by constructor [over.match.ctor]

1 When objects of class type are direct-initialized (8.5), or copy-initialized from an expression of the same or a derived class type (8.5), overload resolution selects the constructor. For direct-initialization, the candidate functions are all the constructors of the class of the object being initialized. For copy-initialization, the candidate functions are all the converting constructors (12.3.1) of that class. The argument list is the *expression-list* within the parentheses of the initializer.

13.3.1.4 Copy-initialization of class by user-defined conversion [over.match.copy]

1 Under the conditions specified in 8.5, as part of a copy-initialization of an object of class type, a user-defined conversion can be invoked to convert an initializer expression to the type of the object being initialized. Overload resolution is used to select the user-defined conversion to be invoked. Assuming that "*cv1* `T`" is the type of the object being initialized, with `T` a class type, the candidate functions are selected as follows:

- The converting constructors (12.3.1) of `T` are candidate functions.
- When the type of the initializer expression is a class type "*cv* `S`", the non-explicit conversion functions of `S` and its base classes are considered. Those that are not hidden within `S` and yield a type whose cv-unqualified version is the same type as `T` or is a derived class thereof are candidate functions. Conversion functions that return "reference to `X`" return lvalues or rvalues, depending on the type of reference, of type `X` and are therefore considered to yield `X` for this process of selecting candidate functions.

2 In both cases, the argument list has one argument, which is the initializer expression. [*Note:* this argument will be compared against the first parameter of the constructors and against the implicit object parameter of the conversion functions. — *end note*]

123) If the value returned by the `operator->` function has class type, this may result in selecting and calling another `operator->` function. The process repeats until an `operator->` function returns a value of non-class type.

13.3.1.5 Initialization by conversion function [over.match.conv]

1 Under the conditions specified in 8.5, as part of an initialization of an object of nonclass type, a conversion function can be invoked to convert an initializer expression of class type to the type of the object being initialized. Overload resolution is used to select the conversion function to be invoked. Assuming that "*cv1* `T`" is the type of the object being initialized, and "*cv* `S`" is the type of the initializer expression, with `S` a class type, the candidate functions are selected as follows:

— The conversion functions of `S` and its base classes are considered. Those non-explicit conversion functions that are not hidden within `S` and yield type `T` or a type that can be converted to type `T` via a standard conversion sequence (13.3.3.1.1) are candidate functions. For direct-initialization, those explicit conversion functions that are not hidden within `S` and yield type `T` or a type that can be converted to type `T` with a qualification conversion (4.4) are also candidate functions. Conversion functions that return a cv-qualified type are considered to yield the cv-unqualified version of that type for this process of selecting candidate functions. Conversion functions that return "reference to *cv2* `X`" return lvalues or rvalues, depending on the type of reference, of type "*cv2* `X`" and are therefore considered to yield `X` for this process of selecting candidate functions.

2 The argument list has one argument, which is the initializer expression. [*Note:* this argument will be compared against the implicit object parameter of the conversion functions. — *end note*]

13.3.1.6 Initialization by conversion function for direct reference binding [over.match.ref]

1 Under the conditions specified in 8.5.3, a reference can be bound directly to an lvalue that is the result of applying a conversion function to an initializer expression. Overload resolution is used to select the conversion function to be invoked. Assuming that "*cv1* `T`" is the underlying type of the reference being initialized, and "*cv* `S`" is the type of the initializer expression, with `S` a class type, the candidate functions are selected as follows:

— The conversion functions of `S` and its base classes are considered, except that for copy-initialization, only the non-explicit conversion functions are considered. Those that are not hidden within `S` and yield type "lvalue reference to *cv2* `T2`", where "*cv1* `T`" is reference-compatible (8.5.3) with "*cv2* `T2`", are candidate functions.

2 The argument list has one argument, which is the initializer expression. [*Note:* this argument will be compared against the implicit object parameter of the conversion functions. — *end note*]

13.3.1.7 Initialization by list-initialization [over.match.list]

1 When objects of non-aggregate class type are list-initialized (8.5.4), overload resolution selects the constructor as follows, where `T` is the cv-unqualified class type of the object being initialized:

— If `T` has an initializer-list constructor (8.5.4), the argument list consists of the initializer list as a single argument; otherwise, the argument list consists of the elements of the initializer list.

— For direct-list-initialization, the candidate functions are all the constructors of the class `T`.

— For copy-list-initialization, the candidate functions are all the constructors of `T`. However, if an `explicit` constructor is chosen, the initialization is ill-formed. [*Note:* This restriction only applies if this initialization is part of the final result of overload resolution — *end note*]

13.3.2 Viable functions [over.match.viable]

1 From the set of candidate functions constructed for a given context (13.3.1), a set of viable functions is chosen, from which the best function will be selected by comparing argument conversion sequences for the best fit (13.3.3). The selection of viable functions considers relationships between arguments and function parameters other than the ranking of conversion sequences.

2 First, to be a viable function, a candidate function shall have enough parameters to agree in number with the arguments in the list.

— If there are m arguments in the list, all candidate functions having exactly m parameters are viable.

— A candidate function having fewer than m parameters is viable only if it has an ellipsis in its parameter list (8.3.5). For the purposes of overload resolution, any argument for which there is no corresponding parameter is considered to "match the ellipsis" (13.3.3.1.3) .

— A candidate function having more than m parameters is viable only if the *(m+1)*-st parameter has a default argument (8.3.6).[124] For the purposes of overload resolution, the parameter list is truncated on the right, so that there are exactly m parameters.

3 Second, for `F` to be a viable function, there shall exist for each argument an *implicit conversion sequence* (13.3.3.1) that converts that argument to the corresponding parameter of `F`. If the parameter has reference type, the implicit conversion sequence includes the operation of binding the reference, and the fact that a reference to non-`const` cannot be bound to an rvalue can affect the viability of the function (see 13.3.3.1.4).

13.3.3 Best Viable Function [over.match.best]

1 Define ICS*i*(`F`) as follows:

— if `F` is a static member function, ICS*1*(`F`) is defined such that ICS*1*(`F`) is neither better nor worse than ICS*1*(`G`) for any function `G`, and, symmetrically, ICS*1*(`G`) is neither better nor worse than ICS*1*(`F`)[125]; otherwise,

— let ICS*i*(`F`) denote the implicit conversion sequence that converts the i-th argument in the list to the type of the i-th parameter of viable function `F`. 13.3.3.1 defines the implicit conversion sequences and 13.3.3.2 defines what it means for one implicit conversion sequence to be a better conversion sequence or worse conversion sequence than another.

Given these definitions, a viable function `F1` is defined to be a *better* function than another viable function `F2` if for all arguments i, ICS*i*(`F1`) is not a worse conversion sequence than ICS*i*(`F2`), and then

— for some argument j, ICS*j*(`F1`) is a better conversion sequence than ICS*j*(`F2`), or, if not that,

— `F1` is a non-template function and `F2` is a function template specialization, or, if not that,

— `F1` and `F2` are function template specializations, and the function template for `F1` is more specialized than the template for `F2` according to the partial ordering rules described in 14.5.6.2, or, if not that,

— the context is an initialization by user-defined conversion (see 8.5, 13.3.1.5, and 13.3.1.6) and the standard conversion sequence from the return type of `F1` to the destination type (i.e., the type of the entity being initialized) is a better conversion sequence than the standard conversion sequence from the return type of `F2` to the destination type. [*Example:*

124) According to 8.3.6, parameters following the *(m+1)*-st parameter must also have default arguments.
125) If a function is a static member function, this definition means that the first argument, the implied object parameter, has no effect in the determination of whether the function is better or worse than any other function.

```
struct A {
  A();
  operator int();
  operator double();
} a;
int i = a;                      // a.operator int() followed by no conversion
                                // is better than a.operator double() followed by
                                // a conversion to int
float x = a;                    // ambiguous: both possibilities require conversions,
                                // and neither is better than the other
```

— *end example*]

2 If there is exactly one viable function that is a better function than all other viable functions, then it is the one selected by overload resolution; otherwise the call is ill-formed[126].

[*Example:*

```
void Fcn(const int*,  short);
void Fcn(int*, int);

int i;
short s = 0;

void f() {
  Fcn(&i, s);                   // is ambiguous because
                                // &i → int* is better than &i → const int*
                                // but s → short is also better than s → int

  Fcn(&i, 1L);                  // calls Fcn(int*, int), because
                                // &i → int* is better than &i → const int*
                                // and 1L → short and 1L → int are indistinguishable

  Fcn(&i,'c');                  // calls Fcn(int*, int), because
                                // &i → int* is better than &i → const int*
                                // and c → int is better than c → short
}
```

— *end example*]

3 If the best viable function resolves to a function for which multiple declarations were found, and if at least two of these declarations — or the declarations they refer to in the case of *using-declaration*s — specify a default argument that made the function viable, the program is ill-formed. [*Example:*

```
namespace A {
  extern "C" void f(int = 5);
}
namespace B {
  extern "C" void f(int = 5);
}
```

126) The algorithm for selecting the best viable function is linear in the number of viable functions. Run a simple tournament to find a function `W` that is not worse than any opponent it faced. Although another function `F` that `W` did not face might be at least as good as `W`, `F` cannot be the best function because at some point in the tournament `F` encountered another function `G` such that `F` was not better than `G`. Hence, `W` is either the best function or there is no best function. So, make a second pass over the viable functions to verify that `W` is better than all other functions.

```
using A::f;
using B::f;

void use() {
  f(3);                          // OK, default argument was not used for viability
  f();                           // Error: found default argument twice
}
```

— *end example*]

13.3.3.1 Implicit conversion sequences [over.best.ics]

1 An *implicit conversion sequence* is a sequence of conversions used to convert an argument in a function call to the type of the corresponding parameter of the function being called. The sequence of conversions is an implicit conversion as defined in Clause 4, which means it is governed by the rules for initialization of an object or reference by a single expression (8.5, 8.5.3).

2 Implicit conversion sequences are concerned only with the type, cv-qualification, and lvalue-ness of the argument and how these are converted to match the corresponding properties of the parameter. Other properties, such as the lifetime, storage class, alignment, or accessibility of the argument and whether or not the argument is a bit-field are ignored. So, although an implicit conversion sequence can be defined for a given argument-parameter pair, the conversion from the argument to the parameter might still be ill-formed in the final analysis.

3 A well-formed implicit conversion sequence is one of the following forms:

— a *standard conversion sequence* (13.3.3.1.1),

— a *user-defined conversion sequence* (13.3.3.1.2), or

— an *ellipsis conversion sequence* (13.3.3.1.3).

4 However, when considering the argument of a user-defined conversion function that is a candidate by 13.3.1.3 when invoked for the copying of the temporary in the second step of a class copy-initialization, by 13.3.1.7 when passing the initializer list as a single argument or when the initializer list has exactly one element and a conversion to some class `X` or reference to (possibly cv-qualified) `X` is considered for the first parameter of a constructor of `X`, or by 13.3.1.4, 13.3.1.5, or 13.3.1.6 in all cases, only standard conversion sequences and ellipsis conversion sequences are allowed.

5 For the case where the parameter type is a reference, see 13.3.3.1.4.

6 When the parameter type is not a reference, the implicit conversion sequence models a copy-initialization of the parameter from the argument expression. The implicit conversion sequence is the one required to convert the argument expression to an rvalue of the type of the parameter. [*Note:* when the parameter has a class type, this is a conceptual conversion defined for the purposes of Clause 13; the actual initialization is defined in terms of constructors and is not a conversion. — *end note*] Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion. [*Example:* a parameter of type `A` can be initialized from an argument of type `const A`. The implicit conversion sequence for that case is the identity sequence; it contains no "conversion" from `const A` to `A`. — *end example*] When the parameter has a class type and the argument expression has the same type, the implicit conversion sequence is an identity conversion. When the parameter has a class type and the argument expression has a derived class type, the implicit conversion sequence is a derived-to-base Conversion from the derived class to the base class. [*Note:* there is no such standard conversion; this derived-to-base Conversion exists only in the description of implicit conversion sequences. — *end note*] A derived-to-base Conversion has Conversion rank (13.3.3.1.1).

7 In all contexts, when converting to the implicit object parameter or when converting to the left operand of an assignment operation only standard conversion sequences that create no temporary object for the result are allowed.

8 If no conversions are required to match an argument to a parameter type, the implicit conversion sequence is the standard conversion sequence consisting of the identity conversion (13.3.3.1.1).

9 If no sequence of conversions can be found to convert an argument to a parameter type or the conversion is otherwise ill-formed, an implicit conversion sequence cannot be formed.

10 If several different sequences of conversions exist that each convert the argument to the parameter type, the implicit conversion sequence associated with the parameter is defined to be the unique conversion sequence designated the *ambiguous conversion sequence*. For the purpose of ranking implicit conversion sequences as described in 13.3.3.2, the ambiguous conversion sequence is treated as a user-defined sequence that is indistinguishable from any other user-defined conversion sequence[127]. If a function that uses the ambiguous conversion sequence is selected as the best viable function, the call will be ill-formed because the conversion of one of the arguments in the call is ambiguous.

11 The three forms of implicit conversion sequences mentioned above are defined in the following subclauses.

13.3.3.1.1 Standard conversion sequences [over.ics.scs]

1 Table 11 summarizes the conversions defined in Clause 4 and partitions them into four disjoint categories: Lvalue Transformation, Qualification Adjustment, Promotion, and Conversion. [*Note:* these categories are orthogonal with respect to lvalue-ness, cv-qualification, and data representation: the Lvalue Transformations do not change the cv-qualification or data representation of the type; the Qualification Adjustments do not change the lvalue-ness or data representation of the type; and the Promotions and Conversions do not change the lvalue-ness or cv-qualification of the type. — *end note*]

2 [*Note:* As described in Clause 4, a standard conversion sequence is either the Identity conversion by itself (that is, no conversion) or consists of one to three conversions from the other four categories. At most one conversion from each category is allowed in a single standard conversion sequence. If there are two or more conversions in the sequence, the conversions are applied in the canonical order: **Lvalue Transformation**, **Promotion** or **Conversion**, **Qualification Adjustment**. — *end note*]

3 Each conversion in Table 11 also has an associated rank (Exact Match, Promotion, or Conversion). These are used to rank standard conversion sequences (13.3.3.2). The rank of a conversion sequence is determined by

127) The ambiguous conversion sequence is ranked with user-defined conversion sequences because multiple conversion sequences for an argument can exist only if they involve different user-defined conversions. The ambiguous conversion sequence is indistinguishable from any other user-defined conversion sequence because it represents at least two user-defined conversion sequences, each with a different user-defined conversion, and any other user-defined conversion sequence must be indistinguishable from at least one of them.

This rule prevents a function from becoming non-viable because of an ambiguous conversion sequence for one of its parameters. Consider this example,

```
class B;
class A { A (B&);};
class B { operator A (); };
class C { C (B&); };
void f(A) { }
void f(C) { }
B b;
f(b);                           // ambiguous because b → C via constructor and
                                // b → A via constructor or conversion function.
```

If it were not for this rule, `f(A)` would be eliminated as a viable function for the call `f(b)` causing overload resolution to select `f(C)` as the function to call even though it is not clearly the best choice. On the other hand, if an `f(B)` were to be declared then `f(b)` would resolve to that `f(B)` because the exact match with `f(B)` is better than any of the sequences required to match `f(A)`.

considering the rank of each conversion in the sequence and the rank of any reference binding (13.3.3.1.4). If any of those has Conversion rank, the sequence has Conversion rank; otherwise, if any of those has Promotion rank, the sequence has Promotion rank; otherwise, the sequence has Exact Match rank.

Table 11 — Conversions

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		4.1
Array-to-pointer conversion			4.2
Function-to-pointer conversion			4.3
Qualification conversions	Qualification Adjustment		4.4
Integral promotions	Promotion	Promotion	4.5
Floating point promotion			4.6
Integral conversions	Conversion	Conversion	4.7
Floating point conversions			4.8
Floating-integral conversions			4.9
Pointer conversions			4.10
Pointer to member conversions			4.11
Boolean conversions			4.12

13.3.3.1.2 User-defined conversion sequences [over.ics.user]

1 A user-defined conversion sequence consists of an initial standard conversion sequence followed by a user-defined conversion (12.3) followed by a second standard conversion sequence. If the user-defined conversion is specified by a constructor (12.3.1), the initial standard conversion sequence converts the source type to the type required by the argument of the constructor. If the user-defined conversion is specified by a conversion function (12.3.2), the initial standard conversion sequence converts the source type to the implicit object parameter of the conversion function.

2 The second standard conversion sequence converts the result of the user-defined conversion to the target type for the sequence. Since an implicit conversion sequence is an initialization, the special rules for initialization by user-defined conversion apply when selecting the best user-defined conversion for a user-defined conversion sequence (see 13.3.3 and 13.3.3.1).

3 If the user-defined conversion is specified by a specialization of a conversion function template, the second standard conversion sequence shall have exact match rank.

4 A conversion of an expression of class type to the same class type is given Exact Match rank, and a conversion of an expression of class type to a base class of that type is given Conversion rank, in spite of the fact that a copy constructor (i.e., a user-defined conversion function) is called for those cases.

13.3.3.1.3 Ellipsis conversion sequences [over.ics.ellipsis]

1 An ellipsis conversion sequence occurs when an argument in a function call is matched with the ellipsis parameter specification of the function called (see 5.2.2).

13.3.3.1.4 Reference binding [over.ics.ref]

1 When a parameter of reference type binds directly (8.5.3) to an argument expression, the implicit conversion sequence is the identity conversion, unless the argument expression has a type that is a derived class of the parameter type, in which case the implicit conversion sequence is a derived-to-base Conversion (13.3.3.1). [*Example:*

```
struct A {};
struct B : public A {} b;
int f(A&);
int f(B&);
int i = f(b);                   // calls f(B&), an exact match, rather than
                                // f(A&), a conversion
```

— *end example*] If the parameter binds directly to the result of applying a conversion function to the argument expression, the implicit conversion sequence is a user-defined conversion sequence (13.3.3.1.2), with the second standard conversion sequence either an identity conversion or, if the conversion function returns an entity of a type that is a derived class of the parameter type, a derived-to-base Conversion.

2 When a parameter of reference type is not bound directly to an argument expression, the conversion sequence is the one required to convert the argument expression to the underlying type of the reference according to 13.3.3.1. Conceptually, this conversion sequence corresponds to copy-initializing a temporary of the underlying type with the argument expression. Any difference in top-level cv-qualification is subsumed by the initialization itself and does not constitute a conversion.

3 A standard conversion sequence cannot be formed if it requires binding an lvalue reference to non-`const` to an rvalue (except when binding an implicit object parameter; see the special rules for that case in 13.3.1). [*Note:* this means, for example, that a candidate function cannot be a viable function if it has a non-`const` lvalue reference parameter (other than the implicit object parameter) and the corresponding argument is a temporary or would require one to be created to initialize the lvalue reference (see 8.5.3). — *end note*]

4 Other restrictions on binding a reference to a particular argument that are not based on the types of the reference and the argument do not affect the formation of a standard conversion sequence, however. [*Example:* a function with an "lvalue reference to `int`" parameter can be a viable candidate even if the corresponding argument is an `int` bit-field. The formation of implicit conversion sequences treats the `int` bit-field as an `int` lvalue and finds an exact match with the parameter. If the function is selected by overload resolution, the call will nonetheless be ill-formed because of the prohibition on binding a non-`const` lvalue reference to a bit-field (8.5.3). — *end example*]

5 The binding of a reference to an expression that is *reference-compatible with added qualification* influences the rank of a standard conversion; see 13.3.3.2 and 8.5.3.

13.3.3.1.5 List-initialization sequence [over.ics.list]

1 When an argument is an initializer list (8.5.4), it is not an expression and special rules apply for converting it to a parameter type.

2 If the parameter type is `std::initializer_list<X>` and all the elements of the initializer list can be implicitly converted to `X`, the implicit conversion sequence is the worst conversion necessary to convert an element of the list to `X`. This conversion can be a user-defined conversion even in the context of a call to an initializer-list constructor. [*Example:*

```
void f(std::initializer_list<int>);
f( {1,2,3} );               // OK: f(initializer_list<int>) identity conversion
f( {'a','b'} );             // OK: f(initializer_list<int>) integral promotion
f( {1.0} );                 // error: narrowing

struct A {
  A(std::initializer_list<double>);           // #1
  A(std::initializer_list<complex<double>>);  // #2
  A(std::initializer_list<std::string>);      // #3
};
A a{ 1.0,2.0 };             // OK, uses #1
```

```
void g(A);
g({ "foo", "bar" });        // OK, uses #3
```

— *end example*]

3 Otherwise, if the parameter is a non-aggregate class X and overload resolution per 13.3.1.7 chooses a single best constructor of X to perform the initialization of an object of type X from the argument initializer list, the implicit conversion sequence is a user-defined conversion sequence. If multiple constructors are viable but none is better than the others, the implicit conversion sequence is the ambiguous conversion sequence. User-defined conversions are allowed for conversion of the initializer list elements to the constructor parameter types except as noted in 13.3.3.1. [*Example:*

```
struct A {
  A(std::initializer_list<int>);
};
void f(A);
f( {'a', 'b'} );            // OK: f(A(std::initializer_list<int>)) user-defined conversion

struct B {
  B(int, double);
};
void g(B);
g( {'a', 'b'} );            // OK: g(B(int,double)) user-defined conversion
g( {1.0, 1,0} );            // error: narrowing

void f(B);
f( {'a', 'b'} );            // error: ambiguous f(A) or f(B)

struct C {
  C(std::string);
};
void h(C);
h({"foo"});                 // OK: h(C(std::string("foo")))

struct D {
  C(A, C);
};
void i(D);
i({ {1,2}, {"bar"} });      // OK: i(D(A(std::initializer_list<int>{1,2}),C(std::string("bar"))))
```

— *end example*]

4 Otherwise, if the parameter has an aggregate type which can be initialized from the initializer list according to the rules for aggregate initialization (8.5.1), the implicit conversion sequence is a user-defined conversion sequence. [*Example:*

```
struct A {
  int m1;
  double m2;
};

void f(A);
f( {'a', 'b'} );            // OK: f(A(int,double)) user-defined conversion
f( {1.0} );                 // error: narrowing
```

— *end example*]

5 Otherwise, if the parameter is a reference, see 13.3.3.1.4. [*Note:* The rules in this section will apply for initializing the underlying temporary for the reference. — *end note*] [*Example:*

```
struct A {
  int m1;
  double m2;
};

void f(const A&);
f( {'a', 'b'} );            // OK: f(A(int,double)) user-defined conversion
f( {1.0} );                 // error: narrowing

void g(const double &);
g({1});                     // same conversion as int to double
```

— *end example*]

6 Otherwise, if the parameter type is not a class:

— if the initializer list has one element, the implicit conversion sequence is the one required to convert the element to the parameter type; [*Example:*

```
void f(int);
f( {'a'} );                 // OK: same conversion as char to int
f( {1.0} );                 // error: narrowing
```

— *end example*]

— if the initializer list has no elements, the implicit conversion sequence is the identity conversion. [*Example:*

```
void f(int);
f( { } );                   // OK: identity conversion
```

— *end example*]

7 In all cases other than those enumerated above, no conversion is possible.

13.3.3.2 Ranking implicit conversion sequences [over.ics.rank]

1 13.3.3.2 defines a partial ordering of implicit conversion sequences based on the relationships *better conversion sequence* and *better conversion*. If an implicit conversion sequence S1 is defined by these rules to be a better conversion sequence than S2, then it is also the case that S2 is a *worse conversion sequence* than S1. If conversion sequence S1 is neither better than nor worse than conversion sequence S2, S1 and S2 are said to be *indistinguishable conversion sequences*.

2 When comparing the basic forms of implicit conversion sequences (as defined in 13.3.3.1)

— a standard conversion sequence (13.3.3.1.1) is a better conversion sequence than a user-defined conversion sequence or an ellipsis conversion sequence, and

— a user-defined conversion sequence (13.3.3.1.2) is a better conversion sequence than an ellipsis conversion sequence (13.3.3.1.3).

3 Two implicit conversion sequences of the same form are indistinguishable conversion sequences unless one of the following rules applies:

— Standard conversion sequence S1 is a better conversion sequence than standard conversion sequence S2 if

— S1 is a proper subsequence of S2 (comparing the conversion sequences in the canonical form defined by 13.3.3.1.1, excluding any Lvalue Transformation; the identity conversion sequence is considered to be a subsequence of any non-identity conversion sequence) or, if not that,

— the rank of S1 is better than the rank of S2, or S1 and S2 have the same rank and are distinguishable by the rules in the paragraph below, or, if not that,

— S1 and S2 differ only in their qualification conversion and yield similar types T1 and T2 (4.4), respectively, and the cv-qualification signature of type T1 is a proper subset of the cv-qualification signature of type T2, and S1 is not the deprecated string literal array-to-pointer conversion (4.2). [*Example:*

```
int f(const int *);
int f(int *);
int i;
int j = f(&i);                  // calls f(int*)
```

— *end example*] or, if not that,

— S1 and S2 are reference bindings (8.5.3) and neither refers to an implicit object parameter of a non-static member function declared without a *ref-qualifier*, and either S1 binds an lvalue reference to an lvalue and S2 binds an rvalue reference or S1 binds an rvalue reference to an rvalue and S2 binds an lvalue reference.

[*Example:*

```
int i;
int f();
int g(const int&);
int g(const int&&);
int j = g(i);                   // calls g(const int&)
int k = g(f());                 // calls g(const int&&)

struct A {
  A& operator<<(int);
  void p() &;
  void p() &&;
};
A& operator<<(A&&, char);
A() << 1;                       // calls A::operator<<(int)
A() << 'c';                     // calls operator<<(A&&, char)
A a;
a << 1;                         // calls A::operator<<(int)
a << 'c';                       // calls operator<<(A&&, char)
A().p();                        // calls A::p()&&
a.p();                          // calls A::p()&
```

— *end example*] or, if not that,

— S1 and S2 are reference bindings (8.5.3), and the types to which the references refer are the same type except for top-level cv-qualifiers, and the type to which the reference initialized by S2 refers is more cv-qualified than the type to which the reference initialized by S1 refers. [*Example:*

```
int f(const int &);
```

```
int f(int &);
int g(const int &);
int g(int);

int i;
int j = f(i);                   // calls f(int &)
int k = g(i);                   // ambiguous

struct X {
  void f() const;
  void f();
};
void g(const X& a, X b) {
  a.f();                        // calls X::f() const
  b.f();                        // calls X::f()
}
```

— *end example*]

— User-defined conversion sequence `U1` is a better conversion sequence than another user-defined conversion sequence `U2` if they contain the same user-defined conversion function or constructor and if the second standard conversion sequence of `U1` is better than the second standard conversion sequence of `U2`. [*Example:*

```
struct A {
  operator short();
} a;
int f(int);
int f(float);
int i = f(a);                   // calls f(int), because short → int is
                                // better than short → float.
```

— *end example*]

4 Standard conversion sequences are ordered by their ranks: an Exact Match is a better conversion than a Promotion, which is a better conversion than a Conversion. Two conversion sequences with the same rank are indistinguishable unless one of the following rules applies:

— A conversion that is not a conversion of a pointer, or pointer to member, to `bool` is better than another conversion that is such a conversion.

— If class `B` is derived directly or indirectly from class `A`, conversion of `B*` to `A*` is better than conversion of `B*` to `void*`, and conversion of `A*` to `void*` is better than conversion of `B*` to `void*`.

— If class `B` is derived directly or indirectly from class `A` and class `C` is derived directly or indirectly from `B`,

— conversion of `C*` to `B*` is better than conversion of `C*` to `A*`, [*Example:*

```
struct A {};
struct B : public A {};
struct C : public B {};
C *pc;
int f(A *);
int f(B *);
int i = f(pc);                  // calls f(B*)
```

— *end example*]

— binding of an expression of type `C` to a reference of type `B&` is better than binding an expression of type `C` to a reference of type `A&`,

— conversion of `A::*` to `B::*` is better than conversion of `A::*` to `C::*`,

— conversion of `C` to `B` is better than conversion of `C` to `A`,

— conversion of `B*` to `A*` is better than conversion of `C*` to `A*`,

— binding of an expression of type `B` to a reference of type `A&` is better than binding an expression of type `C` to a reference of type `A&`,

— conversion of `B::*` to `C::*` is better than conversion of `A::*` to `C::*`, and

— conversion of `B` to `A` is better than conversion of `C` to `A`.

[*Note:* compared conversion sequences will have different source types only in the context of comparing the second standard conversion sequence of an initialization by user-defined conversion (see 13.3.3); in all other contexts, the source types will be the same and the target types will be different. — *end note*]

13.4 Address of overloaded function [over.over]

1 A use of an overloaded function name without arguments is resolved in certain contexts to a function, a pointer to function or a pointer to member function for a specific function from the overload set. A function template name is considered to name a set of overloaded functions in such contexts. The function selected is the one whose type matches the target type required in the context. The target can be

— an object or reference being initialized (8.5, 8.5.3),

— the left side of an assignment (5.17),

— a parameter of a function (5.2.2),

— a parameter of a user-defined operator (13.5),

— the return value of a function, operator function, or conversion (6.6.3),

— an explicit type conversion (5.2.3, 5.2.9, 5.4), or

— a non-type *template-parameter* (14.3.2).

The overloaded function name can be preceded by the `&` operator. An overloaded function name shall not be used without arguments in contexts other than those listed. [*Note:* any redundant set of parentheses surrounding the overloaded function name is ignored (5.1). — *end note*]

2 If the name is a function template, template argument deduction is done (14.8.2.2), and if the argument deduction succeeds, the resulting template argument list is used to generate a single function template specialization, which is added to the set of overloaded functions considered. [*Note:* As described in 14.8.1, if deduction fails and the function template name is followed by an explicit template argument list, the *template-id* is then examined to see whether it identifies a single function template specialization. If it does, the *template-id* is considered to be an lvalue for that function template specialization. The target type is not used in that determination. — *end note*]

3 Non-member functions and static member functions match targets of type "pointer-to-function" or "reference-to-function." Nonstatic member functions match targets of type "pointer-to-member-function;" the function type of the pointer to member is used to select the member function from the set of overloaded member functions. If a non-static member function is selected, the reference to the overloaded function name is required to have the form of a pointer to member as described in 5.3.1.

4 If more than one function is selected, any function template specializations in the set are eliminated if the set also contains a non-template function, and any given function template specialization `F1` is eliminated if the set contains a second function template specialization whose function template is more specialized than the function template of `F1` according to the partial ordering rules of 14.5.6.2. After such eliminations, if any, there shall remain exactly one selected function.

5 [*Example:*

```
int f(double);
int f(int);
int (*pfd)(double) = &f;         // selects f(double)
int (*pfi)(int) = &f;            // selects f(int)
int (*pfe)(...) = &f;            // error: type mismatch
int (&rfi)(int) = f;             // selects f(int)
int (&rfd)(double) = f;          // selects f(double)
void g() {
  (int (*)(int))&f;              // cast expression as selector
}
```

The initialization of `pfe` is ill-formed because no `f()` with type `int(...)` has been declared, and not because of any ambiguity. For another example,

```
struct X {
  int f(int);
  static int f(long);
};

int (X::*p1)(int)  = &X::f;      // OK
int    (*p2)(int)  = &X::f;      // error: mismatch
int    (*p3)(long) = &X::f;      // OK
int (X::*p4)(long) = &X::f;      // error: mismatch
int (X::*p5)(int)  = &(X::f);    // error: wrong syntax for
                                 // pointer to member
int    (*p6)(long) = &(X::f);    // OK
```

— *end example*]

6 [*Note:* if `f()` and `g()` are both overloaded functions, the cross product of possibilities must be considered to resolve `f(&g)`, or the equivalent expression `f(g)`. — *end note*]

7 [*Note:* there are no standard conversions (Clause 4) of one pointer-to-function type into another. In particular, even if `B` is a public base of `D`, we have

```
D* f();
B* (*p1)() = &f;                 // error

void g(D*);
void (*p2)(B*) = &g;             // error
```

— *end note*]

13.5 Overloaded operators [over.oper]

1 A function declaration having one of the following *operator-function-id*s as its name declares an *operator function*. A function template declaration having one of the following *operator-function-id*s as its name

declares an *operator function template*. A specialization of an operator function template is also an operator function. An operator function is said to *implement* the operator named in its *operator-function-id*.

operator-function-id:
`operator` *operator*

operator: one of

`new`	`delete`	`new[]`	`delete[]`					
`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`
`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`	`<<=`	`==`	`!=`
`<=`	`>=`	`&&`	`||`	`++`	`--`	`,`	`->*`	`->`
`()`	`[]`							

[*Note:* the last two operators are function call (5.2.2) and subscripting (5.2.1). The operators `new[]`, `delete[]`, `()`, and `[]` are formed from more than one token. — *end note*]

2 Both the unary and binary forms of

```
+    -    *    &
```

can be overloaded.

3 The following operators cannot be overloaded:

```
.    .*   ::   ?:
```

nor can the preprocessing symbols # and ## (Clause 16).

4 Operator functions are usually not called directly; instead they are invoked to evaluate the operators they implement (13.5.1 – 13.5.7). They can be explicitly called, however, using the *operator-function-id* as the name of the function in the function call syntax (5.2.2). [*Example:*

```
complex z = a.operator+(b);      // complex z = a+b;
void* p = operator new(sizeof(int)*n);
```

— *end example*]

5 The allocation and deallocation functions, `operator new`, `operator new[]`, `operator delete` and `operator delete[]`, are described completely in 3.7.4. The attributes and restrictions found in the rest of this subclause do not apply to them unless explicitly stated in 3.7.4.

6 An operator function shall either be a non-static member function or be a non-member function and have at least one parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration. It is not possible to change the precedence, grouping, or number of operands of operators. The meaning of the operators `=`, (unary) `&`, and `,` (comma), predefined for each type, can be changed for specific class and enumeration types by defining operator functions that implement these operators. Operator functions are inherited in the same manner as other base class functions.

7 The identities among certain predefined operators applied to basic types (for example, `++a` ≡ `a+=1`) need not hold for operator functions. Some predefined operators, such as `+=`, require an operand to be an lvalue when applied to basic types; this is not required by operator functions.

8 An operator function cannot have default arguments (8.3.6), except where explicitly stated below. Operator functions cannot have more or fewer parameters than the number required for the corresponding operator, as described in the rest of this subclause.

9 Operators not mentioned explicitly in subclauses 13.5.3 through 13.5.7 act as ordinary unary and binary operators obeying the rules of 13.5.1 or 13.5.2.

13.5.1 Unary operators [over.unary]

1 A prefix unary operator shall be implemented by a non-static member function (9.3) with no parameters or a non-member function with one parameter. Thus, for any prefix unary operator `@`, `@x` can be interpreted as either `x.operator@()` or `operator@(x)`. If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used. See 13.5.7 for an explanation of the postfix unary operators `++` and `--`.

2 The unary and binary forms of the same operator are considered to have the same name. [*Note:* consequently, a unary operator can hide a binary operator from an enclosing scope, and vice versa. — *end note*]

13.5.2 Binary operators [over.binary]

1 A binary operator shall be implemented either by a non-static member function (9.3) with one parameter or by a non-member function with two parameters. Thus, for any binary operator `@`, `x@y` can be interpreted as either `x.operator@(y)` or `operator@(x,y)`. If both forms of the operator function have been declared, the rules in 13.3.1.2 determine which, if any, interpretation is used.

13.5.3 Assignment [over.ass]

1 An assignment operator shall be implemented by a non-static member function with exactly one parameter. Because a copy assignment operator `operator=` is implicitly declared for a class if not declared by the user (12.8), a base class assignment operator is always hidden by the copy assignment operator of the derived class.

2 Any assignment operator, even the copy assignment operator, can be virtual. [*Note:* for a derived class `D` with a base class `B` for which a virtual copy assignment has been declared, the copy assignment operator in `D` does not override `B`'s virtual copy assignment operator. [*Example:*

```
struct B {
  virtual int operator= (int);
  virtual B& operator= (const B&);
};
struct D : B {
  virtual int operator= (int);
  virtual D& operator= (const B&);
};

D dobj1;
D dobj2;
B* bptr = &dobj1;
void f() {
  bptr->operator=(99);          // calls D::operator=(int)
  *bptr = 99;                   // ditto
  bptr->operator=(dobj2);       // calls D::operator=(const B&)
  *bptr = dobj2;                // ditto
  dobj1 = dobj2;                // calls implicitly-declared
                                // D::operator=(const D&)
}
```

— end example] *— end note*]

13.5.4 Function call [over.call]

1 If declared in a class type, `operator()` shall be a non-static member function with an arbitrary number of parameters. It can have default arguments. It implements the function call syntax

postfix-expression `(` *expression-list*$_{opt}$ `)`

where the *postfix-expression* evaluates to a class object and the possibly empty *expression-list* matches the parameter list of an `operator()` member function of the class. Thus, a call `x(arg1,...)` is interpreted as `x.operator()(arg1, ...)` for a class object `x` of type `T` if `T::operator()(T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

2 If declared in a concept or concept map, `operator()` shall be a non-member associated function with one or more parameters. It implements the function call syntax

postfix-expression `(` *expression-list*$_{opt}$ `)`

where the *postfix-expression* evaluates to an object and the possibly empty *expression-list* matches the parameter list of the `operator()` associated function after the first parameter of the parameter list has been removed. Thus, a call `x(arg1,...)` is interpreted as `operator()(x, arg1, ...)` for an object `x` of type `T` if `operator()(T, T1, T2, T3)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

13.5.5 Subscripting [over.sub]

1 If declared in a class type, `operator[]` shall be a non-static member function with exactly one parameter. It implements the subscripting syntax

postfix-expression `[` *expression* `]`

Thus, a subscripting expression `x[y]` is interpreted as `x.operator[](y)` for a class object `x` of type `T` if `T::operator[](T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

2 If declared in a concept or concept map, `operator[]` shall be a non-member associated function with exactly two parameters. It implements the subscripting syntax

postfix-expression `[` *expression* `]`

Thus, a subscripting expression `x[y]` is interpreted as `operator[](x, y)` for an object `x` of type `T` if `operator[](T, T1)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3.3).

13.5.6 Class member access [over.ref]

1 If declared in a class type, `operator->` shall be a non-static member function taking no parameters. It implements the class member access syntax that uses `->`.

postfix-expression `->` `template`$_{opt}$ *id-expression*
postfix-expression `->` *pseudo-destructor-name*

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3).

2 If declared in a concept or concept map, `operator->` shall be a non-member associated function taking exactly one parameter. It implements class member access using `->`

postfix-expression `->` *id-expression*

An expression `x->m` is interpreted as `(operator->(x))->m` for an object `x` of type `T` if `operator->(T)` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3).

13.5.7 Increment and decrement [over.inc]

1 The user-defined function called `operator++` implements the prefix and postfix `++` operator. If this function is a member function with no parameters, or a non-member function with one parameter of class or enumeration type, it defines the prefix increment operator `++` for objects of that type. If the function is a member function with one parameter (which shall be of type `int`) or a non-member function with two parameters (the second of which shall be of type `int`), it defines the postfix increment operator `++` for objects of that type. When the postfix increment is called as a result of using the `++` operator, the `int` argument will have value zero.[128] [*Example:*

```
struct X {
  X&   operator++();              // prefix ++a
  X    operator++(int);           // postfix a++
};

struct Y { };
Y&   operator++(Y&);              // prefix ++b
Y    operator++(Y&, int);         // postfix b++

void f(X a, Y b) {
  ++a;                            // a.operator++();
  a++;                            // a.operator++(0);
  ++b;                            // operator++(b);
  b++;                            // operator++(b, 0);

  a.operator++();                 // explicit call: like ++a;
  a.operator++(0);                // explicit call: like a++;
  operator++(b);                  // explicit call: like ++b;
  operator++(b, 0);               // explicit call: like b++;
}
```

— *end example*]

2 The prefix and postfix decrement operators `--` are handled analogously.

13.5.8 User-defined literals [over.literal]

literal-operator-id:
`operator ""` *identifier*

1 The *identifier* in a *literal-operator-id* is called a *literal suffix identifier*.

2 A declaration whose *declarator-id* is a *literal-operator-id* shall be a declaration of a namespace-scope function or function template (it could be a friend function (11.4)), an explicit instantiation or specialization of a function template, or a *using-declaration* (7.3.3). A function declared with a *literal-operator-id* is a *literal operator*. A function template declared with a *literal-operator-id* is a *literal operator template*.

3 The declaration of a literal operator shall have a *parameter-declaration-clause* equivalent to one of the following:

```
const char*
unsigned long long int
```

128) Calling `operator++` explicitly, as in expressions like `a.operator++(2)`, has no special properties: The argument to `operator++` is 2.

```
long double
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

4 A *raw literal operator* is a literal operator with a single parameter whose type is *const char**.

5 The declaration of a literal operator template shal have an empty *parameter-declaration-clause* and its *template-parameter-list* shall have a single *template-parameter* that is a non-type template parameter pack with element type `char`.

6 Literal operators and literal operator templates shall not have C language linkage.

7 [*Note:* literal operators and literal operator templates are usually invoked implicitly through user-defined literals (2.13.7). However, except for the constraints described above, they are ordinary namespace-scope functions and function templates. In particular, they are looked up like ordinary functions and function templates and they follow the same overload resolution rules. Also, they can be declared `inline` or `constexpr`, they may have internal or external linkage, they can be called explicitly, their addresses can be taken, etc. — *end note*]

8 [*Example:*

```
void operator "" _km(long double);                  // OK
string operator "" _i18n(const char*, std::size_t); // OK
template <char...> int operator "" \u03C0();        // OK: UCN for lowercase pi
float operator ""E(const char*);                    // error: ""E (with no intervening space)
                                                    // is a single token
float operator " " B(const char*);                  // error: non-adjacent quotes
string operator "" 5X(const char*, std::size_t);    // error: invalid lteral suffix identifier
double operator _miles(double);                     // error: invalid parameter-declaration-clause
template <char...> int operator "" j(const char*);  // error: invalid parameter-declaration-clause
```

— *end example*]

13.6 Built-in operators [over.built]

1 The candidate operator functions that represent the built-in operators defined in Clause 5 are specified in this subclause. These candidate functions participate in the operator overload resolution process as described in 13.3.1.2 and are used for no other purpose. Built-in operators are not defined for archetypes (14.10.2), except as noted, even though template requirements naming compiler-supported concepts (14.9.4) can classify archetypes as non-class types. [*Note:* because built-in operators take only operands with non-class type, and operator overload resolution occurs only when an operand expression originally has class or enumeration type, operator overload resolution can resolve to a built-in operator only when an operand has a class type that has a user-defined conversion to a non-class type appropriate for the operator, or when an operand has an enumeration type that can be converted to a type appropriate for the operator. Also note that some of the candidate operator functions given in this subclause are more permissive than the built-in operators themselves. As described in 13.3.1.2, after a built-in operator is selected by overload resolution the expression is subject to the requirements for the built-in operator given in Clause 5, and therefore to any additional semantic constraints given there. If there is a user-written candidate with the same name and parameter types as a built-in candidate operator function, the built-in operator function is hidden and is not included in the set of candidate functions. — *end note*]

2 In this subclause, the term *promoted integral type* is used to refer to those integral types which are preserved by integral promotion (including e.g. `int` and `long` but excluding e.g. `char`). Similarly, the term *promoted*

arithmetic type refers to floating types plus promoted integral types. [*Note:* in all cases where a promoted integral type or promoted arithmetic type is required, an operand of enumeration type will be acceptable by way of the integral promotions. — *end note*]

3 For every pair (*T*, *VQ*), where *T* is an arithmetic type, and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T& operator++(VQ T&);
T operator++(VQ T&, int);
```

4 For every pair (*T*, *VQ*), where *T* is an arithmetic type other than *bool*, and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T& operator--(VQ T&);
T operator--(VQ T&, int);
```

5 For every pair (*T*, *VQ*), where *T* is a cv-qualified or cv-unqualified object type, and *VQ* is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ& operator++(T*VQ&);
T*VQ& operator--(T*VQ&);
T*    operator++(T*VQ&, int);
T*    operator--(T*VQ&, int);
```

6 For every cv-qualified or cv-unqualified effective object type *T*, there exist candidate operator functions of the form

```
T&    operator*(T*);
```

7 For every function type *T*, there exist candidate operator functions of the form

```
T&    operator*(T*);
```

8 For every type *T*, including archetypes for which the template requirements contain `std::PointeeType<T>`, there exist candidate operator functions of the form

```
T*    operator+(T*);
```

9 For every promoted arithmetic type *T*, there exist candidate operator functions of the form

```
T operator+(T);
T operator-(T);
```

10 For every promoted integral type *T*, there exist candidate operator functions of the form

```
T operator~(T);
```

11 For every quintuple (*C1*, *C2*, *T*, *CV1*, *CV2*), where *C2* is a class type, *C1* is the same type as C2 or is a derived class of C2, *T* is an effective object type or a function type, and *CV1* and *CV2* are *cv-qualifier-seq*s, there exist candidate operator functions of the form

```
CV12 T& operator->*(CV1 C1*, CV2 T C2::*);
```

where *CV12* is the union of *CV1* and *CV2*.

12 For every pair of promoted arithmetic types *L* and *R*, there exist candidate operator functions of the form

```
LR      operator*(L, R);
LR      operator/(L, R);
LR      operator+(L, R);
LR      operator-(L, R);
bool    operator<(L, R);
bool    operator>(L, R);
bool    operator<=(L, R);
bool    operator>=(L, R);
bool    operator==(L, R);
bool    operator!=(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

13 For every cv-qualified or cv-unqualified effective object type *T* there exist candidate operator functions of the form

```
T*      operator+(T*, std::ptrdiff_t);
T&      operator[](T*, std::ptrdiff_t);
T*      operator-(T*, std::ptrdiff_t);
T*      operator+(std::ptrdiff_t, T*);
T&      operator[](std::ptrdiff_t, T*);
```

14 For every *T*, where *T* is a pointer to effective object type, there exist candidate operator functions of the form

```
std::ptrdiff_t   operator-(T, T);
```

15 For every *T*, where *T* is an enumeration type or pointer to effective object type, there exist candidate operator functions of the form

```
bool    operator<(T, T);
bool    operator>(T, T);
bool    operator<=(T, T);
bool    operator>=(T, T);
bool    operator==(T, T);
bool    operator!=(T, T);
```

16 For every pointer to member type *T*, including pointer to member types involving archetypes, there exist candidate operator functions of the form

```
bool    operator==(T, T);
bool    operator!=(T, T);
```

17 For every pair of promoted integral types *L* and *R*, there exist candidate operator functions of the form

```
LR      operator%(L, R);
LR      operator&(L, R);
LR      operator^(L, R);
LR      operator|(L, R);
L       operator<<(L, R);
L       operator>>(L, R);
```

where *LR* is the result of the usual arithmetic conversions between types *L* and *R*.

18 For every triple (*L*, *VQ*, *R*), where *L* is an arithmetic type, *VQ* is either `volatile` or empty, and *R* is a promoted arithmetic type, there exist candidate operator functions of the form

```
VQ L&   operator=(VQ L&, R);
VQ L&   operator*=(VQ L&, R);
VQ L&   operator/=(VQ L&, R);
VQ L&   operator+=(VQ L&, R);
VQ L&   operator-=(VQ L&, R);
```

19 For every pair (T, VQ), where T is any type, including archetypes for which the template requirements contain `std::PointeeType<T>`, and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ&   operator=(T*VQ&, T*);
```

20 For every pair (T, VQ), where T is an enumeration or pointer to member type, including pointer to member types that involve archetypes, and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
VQ T&   operator=(VQ T&, T);
```

21 For every pair (T, VQ), where T is a cv-qualified or cv-unqualified effective object type and VQ is either `volatile` or empty, there exist candidate operator functions of the form

```
T*VQ&   operator+=(T*VQ&, std::ptrdiff_t);
T*VQ&   operator-=(T*VQ&, std::ptrdiff_t);
```

22 For every triple (L, VQ, R), where L is an integral type, VQ is either `volatile` or empty, and R is a promoted integral type, there exist candidate operator functions of the form

```
VQ L&   operator%=(VQ L&, R);
VQ L&   operator<<=(VQ L&, R);
VQ L&   operator>>=(VQ L&, R);
VQ L&   operator&=(VQ L&, R);
VQ L&   operator^=(VQ L&, R);
VQ L&   operator|=(VQ L&, R);
```

23 There also exist candidate operator functions of the form

```
bool    operator!(bool);
bool    operator&&(bool, bool);
bool    operator||(bool, bool);
```

24 For every pair of promoted arithmetic types L and R, there exist candidate operator functions of the form

```
LR      operator?(bool, L, R);
```

where LR is the result of the usual arithmetic conversions between types L and R. [*Note:* as with all these descriptions of candidate functions, this declaration serves only to describe the built-in operator for purposes of overload resolution. The operator "`?`" cannot be overloaded. — *end note*]

25 For every type T, where T is a pointer or pointer-to-member type, including pointer and pointer-to-member types that involve archetypes, there exist candidate operator functions of the form

```
T       operator?(bool, T, T);
```

14 Templates [temp]

1 A *template* defines a family of classes, functions, or concept maps, or an alias for a family of types.

template-declaration:
`export`$_{opt}$ `template` `<` *template-parameter-list* `>` *declaration*

template-parameter-list:
template-parameter
template-parameter-list `,` *template-parameter*

[*Note:* The `>` token following the *template-parameter-list* of a *template-declaration* may be the product of replacing a `>>` token by two consecutive `>` tokens (14.2). — *end note*]

The *declaration* in a *template-declaration* shall

— declare or define a function or a class, or

— define a member function, a member class or a static data member of a class template or of a class nested within a class template, or

— define a member template of a class or class template, or

— be an *alias-declaration*, or

— define a concept map.

A *template-declaration* is a *declaration*. A *template-declaration* is also a definition if its *declaration* defines a function, a class, a concept map, or a static data member.

2 A *template-declaration* can appear only as a namespace scope or class scope declaration. In a function template declaration, the last component of the *declarator-id* shall be a *template-name* or *operator-function-id* (i.e., not a *template-id*). [*Note:* in a class template declaration, if the class name is a *simple-template-id*, the declaration declares a class template partial specialization (14.5.5). — *end note*]

3 In a *template-declaration*, explicit specialization, or explicit instantiation the *init-declarator-list* in the declaration shall contain at most one declarator. When such a declaration is used to declare a class template, no declarator is permitted.

4 A template name has linkage (3.5). A non-member function template can have internal linkage; any other template name shall have external linkage. Entities generated from a template with internal linkage are distinct from all entities generated in other translation units. A template, a template explicit specialization (14.7.3), and a class template partial specialization shall not have C linkage. Use of a linkage specification other than C or C++ with any of these constructs is conditionally-supported, with implementation-defined semantics. Template definitions shall obey the one definition rule (3.2). [*Note:* default arguments for function templates and for member functions of class templates are considered definitions for the purpose of template instantiation (14.5) and must also obey the one definition rule. — *end note*]

5 A class template shall not have the same name as any other template, class, concept, function, object, enumeration, enumerator, namespace, or type in the same scope (3.3), except as specified in (14.5.5). Except that a function template can be overloaded either by (non-template) functions with the same name or by other function templates with the same name (14.8.3), a template name declared in namespace scope or in class scope shall be unique in that scope.

6 A *template-declaration* may be preceded by the `export` keyword. Such a template is said to be *exported*. Declaring exported a class template is equivalent to declaring exported all of its non-inline member functions, static data members, member classes, member class templates, and non-inline member function templates.

7 If a template is exported in one translation unit, it shall be exported in all translation units in which it appears; no diagnostic is required. A declaration of an exported template shall appear with the `export` keyword before any point of instantiation (14.6.4.1) of that template in that translation unit. In addition, the first declaration of an exported template containing the `export` keyword shall not follow the definition of that template. The `export` keyword shall not be used in a friend declaration.

8 Templates defined in an unnamed namespace, inline functions, and inline function templates shall not be exported. An exported non-class template shall be defined only once in a program; no diagnostic is required. An exported non-class template need only be declared (and not necessarily defined) in a translation unit in which it is instantiated.

9 A non-exported non-class template shall be defined in every translation unit in which it is implicitly instantiated (14.7.1), unless the corresponding specialization is explicitly instantiated (14.7.2) in some translation unit; no diagnostic is required.

10 [*Note:* an implementation may require that a translation unit containing the definition of an exported template be compiled before any translation unit containing an instantiation of that template. — *end note*]

11 A *template-declaration* that declares a template alias (14.5.7) shall not be exported.

12 A *template-declaration* with a `requires` keyword is a constrained template (14.10). The *requires-clause* specifies template requirements (14.10.1).

14.1 Template parameters [temp.param]

1 The syntax for *template-parameter*s is:

template-parameter:
type-parameter
parameter-declaration
constrained-template-parameter

type-parameter:
`class` ...$_{opt}$ *identifier*$_{opt}$
`class` *identifier*$_{opt}$ `=` *type-id*
`typename` ...$_{opt}$ *identifier*$_{opt}$
`typename` *identifier*$_{opt}$ `=` *type-id*
`template` `<` *template-parameter-list* `>` `class` ...$_{opt}$ *identifier*$_{opt}$
`template` `<` *template-parameter-list* `>` `class` *identifier*$_{opt}$ `=` *id-expression*

constrained-template-parameter:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* ...$_{opt}$ *identifier*$_{opt}$
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name identifier*$_{opt}$ *constrained-default-argument*$_{opt}$
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `<`
simple-requirement-argument-list `>` ...$_{opt}$ *identifier*
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `<`
simple-requirement-argument-list `>` *identifier constrained-default-argument*$_{opt}$

constrained-default-argument:
`=` *type-id*
`=` *assignment-expression*
`=` *id-expression*

simple-requirement-argument-list:
`auto`
`auto ,` *template-argument-list*

[*Note:* The `>` token following the *template-parameter-list* of a *type-parameter* may be the product of replacing a `>>` token by two consecutive `>` tokens (14.2). — *end note*]

2 There is no semantic difference between `class` and `typename` in a *template-parameter*. `typename` followed by an *unqualified-id* names a template type parameter. `typename` followed by a *qualified-id* denotes the type in a non-type [129] *parameter-declaration*. A storage class shall not be specified in a *template-parameter* declaration. [*Note:* a template parameter may be a class template. For example,

```
template<class T> class myarray { /* ... */ };

template<class K, class V, template<class T> class C = myarray>
class Map {
  C<K> key;
  C<V> value;
};
```

— *end note*]

3 A *type-parameter* whose identifier does not follow an ellipsis defines its *identifier* to be a *typedef-name* (if declared with `class` or `typename`) or *template-name* (if declared with `template`) in the scope of the template declaration. [*Note:* because of the name lookup rules, a *template-parameter* that could be interpreted as either a non-type *template-parameter* or a *type-parameter* (because its *identifier* is the name of an already existing class) is taken as a *type-parameter*. For example,

```
class T { /* ... */ };
int i;

template<class T, T i> void f(T t) {
  T t1 = i;          // template-parameters T and i
  ::T t2 = ::i;      // global namespace members T and i
}
```

Here, the template `f` has a *type-parameter* called `T`, rather than an unnamed non-type *template-parameter* of class `T`. — *end note*]

4 A non-type *template-parameter* shall have one of the following (optionally *cv-qualified*) types:

— integral or enumeration type,

— pointer to object or pointer to function,

— reference to object or reference to function,

— pointer to member, or

— in a constrained template (14.10), a type archetype `T` for which the concept requirement `std::NonTypeTemplateParameterType<T>` (14.9.4) is part of the template's requirements.

5 [*Note:* other types are disallowed either explicitly below or implicitly by the rules governing the form of *template-argument*s (14.3). — *end note*] The top-level *cv-qualifiers* on the *template-parameter* are ignored when determining its type.

129) Since template *template-parameter*s and template *template-argument*s are treated as types for descriptive purposes, the terms *non-type parameter* and *non-type argument* are used to refer to non-type, non-template parameters and arguments.

6 A non-type non-reference *template-parameter* is not an lvalue. It shall not be assigned to or in any other way have its value changed. A non-type non-reference *template-parameter* cannot have its address taken. When a non-type non-reference *template-parameter* is used as an initializer for a reference, a temporary is always used. [*Example:*

```
template<const X& x, int i> void f() {
  i++;                          // error: change of template-parameter value

  &x;                           // OK
  &i;                           // error: address of non-reference template-parameter

  int& ri = i;                  // error: non-const reference bound to temporary
  const int& cri = i;           // OK: const reference bound to temporary
}
```

— *end example*]

7 A non-type *template-parameter* shall not be declared to have floating point, class, or void type. [*Example:*

```
template<double d> class X;     // error
template<double* pd> class Y;   // OK
template<double& rd> class Z;   // OK
```

— *end example*]

8 A non-type *template-parameter* of type "array of `T`" or "function returning `T`" is adjusted to be of type "pointer to `T`" or "pointer to function returning `T`", respectively. [*Example:*

```
template<int *a>   struct R { /* ... */ };
template<int b[5]> struct S { /* ... */ };
int p;
R<&p> w;                        // OK
S<&p> x;                        // OK due to parameter adjustment
int v[5];
R<v> y;                         // OK due to implicit argument conversion
S<v> z;                         // OK due to both adjustment and conversion
```

— *end example*]

9 A *default template-argument* is a *template-argument* (14.3) specified after `=` in a *template-parameter*. A default *template-argument* may be specified for any kind of *template-parameter* (type, non-type, template) that is not a template parameter pack. A default *template-argument* may be specified in a template declaration. A default *template-argument* shall not be specified in the *template-parameter-list*s of the definition of a member of a class template that appears outside of the member's class. A default *template-argument* shall not be specified in a friend class template declaration. If a friend function template declaration specifies a default *template-argument*, that declaration shall be a definition and shall be the only declaration of the function template in the translation unit.

10 The set of default *template-argument*s available for use with a template declaration or definition is obtained by merging the default arguments from the definition (if in scope) and all declarations in scope in the same way default function arguments are (8.3.6). [*Example:*

```
template<class T1, class T2 = int> class A;
template<class T1 = int, class T2> class A;
```

is equivalent to

```
template<class T1 = int, class T2 = int> class A;
```

— *end example*]

11 If a *template-parameter* of a class template has a default *template-argument*, each subsequent *template-parameter* shall either have a default *template-argument* supplied or be a template parameter pack. If a *template-parameter* of a class template is a template parameter pack, it shall be the last *template-parameter*. [*Note:* These are not requirements for function templates because template arguments might be deduced (14.8.2). [*Example:*

```
template<class T1 = int, class T2> class B;   // error
```

— *end example*] — *end note*]

12 A *template-parameter* shall not be given default arguments by two different declarations in the same scope. [*Example:*

```
template<class T = int> class X;
template<class T = int> class X { /*... */ }; // error
```

— *end example*]

13 The scope of a *template-parameter* extends from its point of declaration until the end of its template. In particular, a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments. [*Example:*

```
template<class T, T* p, class U = T> class X { /* ... */ };
template<class T> void f(T* p = new T);
```

— *end example*]

14 A *template-parameter* shall not be used in its own default argument.

15 When parsing a default *template-argument* for a non-type *template-parameter*, the first non-nested > is taken as the end of the *template-parameter-list* rather than a greater-than operator. [*Example:*

```
template<int i = 3 > 4 >        // syntax error
class X { /* ... */ };

template<int i = (3 > 4) >      // OK
class Y { /* ... */ };
```

— *end example*]

16 A *template-parameter* of a template *template-parameter* is permitted to have a default *template-argument*. When such default arguments are specified, they apply to the template *template-parameter* in the scope of the template *template-parameter*. [*Example:*

```
template <class T = float> struct B {};
template <template <class TT = float> class T> struct A {
  inline void f();
  inline void g();
};
template <template <class TT> class T> void A<T>::f() {
  T<> t;            // error - TT has no default template argument
}
template <template <class TT = char> class T> void A<T>::g() {
    T<> t;          // OK - T<char>
}
```

— *end example*]

17 If a *template-parameter* is a *type-parameter* with an ellipsis prior to its optional *identifier* or is a *parameter-declaration* that declares a parameter pack (8.3.5), then the *template-parameter* is a template parameter pack (14.5.3). [*Example:*

```
template <class... Types> class Tuple;              // Types is a template type parameter pack
template <class T, int... Dims> struct multi_array; // Dims is a non-type template parameter pack
```

— *end example*]

18 A *template-parameter* declared with a *concept-name* is a template type, non-type or template parameter or parameter pack that specifies a template requirement (14.10.1) using the *simple form* of template requirements. The kind (type, non-type or template) of the parameter is that of the first template parameter of the concept named in the *constrained-template-parameter*. For a non-type parameter, the type is that of the first template parameter of the concept named in the *constrained-template-parameter*. For a template parameter, the template parameter list is that of the first template parameter of the concept named in the *constrained-template-parameter*. A template parameter or parameter pack written $::_{opt}$ *nested-name-specifier*$_{opt}$ *C* $..._{opt}$ *T*, where `C` is a *concept-name*, is equivalent to a template parameter or parameter pack `T` declared as a *type-parameter* or *parameter-declaration* with the template requirement or pack expansion $::_{opt}$ *nested-name-specifier*$_{opt}$ `C<T>` $..._{opt}$ added to the template requirements. A template parameter or parameter pack written $::_{opt}$ *nested-name-specifier*$_{opt}$ `C<auto, T2, T3, ..., T`*N*`>...`$_{opt}$ `T`, is equivalent to a template parameter or parameter pack `T` declared as a *type-parameter* or *parameter-declaration* with the template requirement $::_{opt}$ *nested-name-specifier*$_{opt}$ `C<T, T2, T3, ..., T`*N*`>...`$_{opt}$ added to the template requirements.

```
concept C<typename T> { ... }
concept D<typename T, typename U> { ... }
concept E<typename T, typename U, typename V = U> { ... }

template<C T, D<auto, T> P> void f(T, P);
// equivalent to
template<class T, class P> requires C<T> && D<P, T> void f(T, P);

template<C T, E<auto, T> P> void f(T, P);
// equivalent to
template<class T, class P> requires C<T> && E<P, T, T> void f(T, P);
```

— *end example*]

When the *type-parameter* is a template type parameter pack, the equivalent requirement is a pack expansion (14.5.3). [*Example:*

```
concept C<typename T> { }

template<C... Args> void g(Args const&...);
// equivalent to
template<typename... Args> requires C<Args>... void g(Args const&...);
```

— *end example*]

14.2 Names of template specializations [temp.names]

1 A template specialization (14.7) can be referred to by a *template-id*:

simple-template-id:
template-name < *template-argument-list*$_{opt}$ >

template-id:
simple-template-id
operator-function-id < *template-argument-list*$_{opt}$ >

template-name:
identifier

template-argument-list:
template-argument ...$_{opt}$
template-argument-list , *template-argument* ...$_{opt}$

template-argument:
constant-expression
type-id
id-expression

[*Note:* the name lookup rules (3.4) are used to associate the use of a name with a template declaration; that is, to identify a name as a *template-name*. — *end note*]

2 For a *template-name* to be explicitly qualified by the template arguments, the name must be known to refer to a template.

3 After name lookup (3.4) finds that a name is a *template-name*, or that an *operator-function-id* refers to a set of overloaded functions any member of which is a function template, if this is followed by a <, the < is always taken as the delimiter of a *template-argument-list* and never as the less-than operator. When parsing a *template-argument-list*, the first non-nested >[130] is taken as the ending delimiter rather than a greater-than operator. Similarly, the first non-nested >> is treated as two consecutive but distinct > tokens, the first of which is taken as the end of the *template-argument-list* and completes the *template-id*. [*Note:* The second > token produced by this replacement rule may terminate an enclosing *template-id* construct or it may be part of a different construct (e.g. a cast). — *end note*] [*Example:*

```
template<int i> class X { /* ... */ };

X< 1>2 > x1;         // syntax error
X<(1>2)> x2;         // OK

template<class T> class Y { /* ... */ };
Y<X<1>> x3;          // OK, same as Y<X<1> > x3;
Y<X<6>>1>> x4;       // syntax error
Y<X<(6>>1)>> x5;     // OK
```

— *end example*]

4 When the name of a member template specialization appears after . or -> in a *postfix-expression*, or after a *nested-name-specifier* in a *qualified-id*, and the *postfix-expression* or *qualified-id* explicitly depends on a *template-parameter* (14.6.2) but does not refer to a member of the current instantiation (14.6.2.1), the member template name must be prefixed by the keyword `template`. Otherwise the name is assumed to name a non-template. [*Example:*

```
struct X {
  template<std::size_t> X* alloc();
  template<std::size_t> static X* adjust();
};
```

130) A > that encloses the *type-id* of a `dynamic_cast`, `static_cast`, `reinterpret_cast` or `const_cast`, or which encloses the *template-argument*s of a subsequent *template-id*, is considered nested for the purpose of this description.

```
template<class T> void f(T* p) {
  T* p1 = p->alloc<200>();          // ill-formed: < means less than
  T* p2 = p->template alloc<200>(); // OK: < starts template argument list
  T::adjust<100>();                 // ill-formed: < means less than
  T::template adjust<100>();        // OK: < starts template argument list
}
```

— *end example*]

5 If a name prefixed by the keyword `template` is not the name of a template, the program is ill-formed. [*Note:* the keyword `template` may not be applied to non-template members of class templates. — *end note*] [*Note:* as is the case with the `typename` prefix, the `template` prefix is allowed in cases where it is not strictly necessary; i.e., when the *nested-name-specifier* or the expression on the left of the `->` or `.` is not dependent on a *template-parameter*, or the use does not appear in the scope of a template. — *end note*]

6 A *simple-template-id* that names a class template specialization is a *class-name* (Clause 9).

7 A *template-id* that names a template alias specialization is a *type-name*.

14.3 Template arguments [temp.arg]

1 There are three forms of *template-argument*, corresponding to the three forms of *template-parameter*: type, non-type and template. The type and form of each *template-argument* specified in a *template-id* shall match the type and form specified for the corresponding parameter declared by the template in its *template-parameter-list*. When the parameter declared by the template is a template parameter pack, it will correspond to zero or more *template-argument*s. [*Example:*

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};

Array<int> v1(20);
typedef std::complex<double> dcomplex;  // std::complex is a standard
                                        // library template
Array<dcomplex> v2(30);
Array<dcomplex> v3(40);

void bar() {
  v1[3] = 7;
  v2[3] = v3.elem(4) = dcomplex(7,8);
}
```

— *end example*]

2 In a *template-argument*, an ambiguity between a *type-id* and an expression is resolved to a *type-id*, regardless of the form of the corresponding *template-parameter*.[131] [*Example:*

```
template<class T> void f();
template<int I> void f();
```

131) There is no such ambiguity in a default *template-argument* because the form of the *template-parameter* determines the allowable forms of the *template-argument*.

```
void g() {
  f<int()>();       // int() is a type-id: call the first f()
}
```

— *end example*]

3 The name of a *template-argument* shall be accessible at the point where it is used as a *template-argument*. [*Note:* if the name of the *template-argument* is accessible at the point where it is used as a *template-argument*, there is no further access restriction in the resulting instantiation where the corresponding *template-parameter* name is used. — *end note*] [*Example:*

```
template<class T> class X {
  static T t;
};

class Y {
private:
  struct S { /* ... */ };
  X<S> x;           // OK: S is accessible
                    // X<Y::S> has a static member of type Y::S
                    // OK: even though Y::S is private
};

X<Y::S> y;          // error: S not accessible
```

— *end example*] For a *template-argument* that is a class type or a class template, the template definition has no special access rights to the members of the *template-argument*. [*Example:*

```
template <template <class TT> class T> class A {
  typename T<int>::S s;
};

template <class U> class B {
private:
  struct S { /* ... */ };
};

A<B> b;             // ill-formed: A has no access to B::S
```

— *end example*]

4 When template argument packs or default *template-argument*s are used, a *template-argument* list can be empty. In that case the empty <> brackets shall still be used as the *template-argument-list*. [*Example:*

```
template<class T = char> class String;
String<>* p;                    // OK: String<char>
String* q;                      // syntax error
template<class ... Elements> class Tuple;
Tuple<>* t;                     // OK: Elements is empty
Tuple* u;                       // syntax error
```

— *end example*]

5 An explicit destructor call (12.4) for an object that has a type that is a class template specialization may explicitly specify the *template-argument*s. [*Example:*

```
template<class T> struct A {
  ~A();
};
void f(A<int>* p, A<int>* q) {
  p->A<int>::~A();              // OK: destructor call
  q->A<int>::~A<int>();         // OK: destructor call
}
```

— *end example*]

6 If the use of a *template-argument* gives rise to an ill-formed construct in the instantiation of a template specialization, the program is ill-formed.

7 When the template in a *template-id* is an overloaded function template, both non-template functions in the overload set and function templates in the overload set for which the *template-argument*s do not match the *template-parameter*s are ignored. If none of the function templates have matching *template-parameter*s, the program is ill-formed.

8 A *template-argument* followed by an ellipsis is a pack expansion (14.5.3).

14.3.1 Template type arguments [temp.arg.type]

1 A *template-argument* for a *template-parameter* which is a type shall be a *type-id*.

2 [*Example:*

```
template <class T> class X { };
template <class T> void f(T t) { }
struct { } unnamed_obj;

void f() {
  struct A { };
  enum { e1 };
  typedef struct { } B;
  B b;
  X<A> x1;          // OK
  X<A*> x2;         // OK
  X<B> x3;          // OK
  f(e1);            // OK
  f(unnamed_obj);   // OK
  f(b);             // OK
}
```

— *end example*] [*Note:* a template type argument may be an incomplete type (3.9). — *end note*]

3 If a declaration acquires a function type through a type dependent on a *template-parameter* and this causes a declaration that does not use the syntactic form of a function declarator to have function type, the program is ill-formed. [*Example:*

```
template<class T> struct A {
  static T t;
};
typedef int function();
A<function> a;                  // ill-formed: would declare A<function>::t
                                // as a static member function
```

— *end example*]

4 If a *template-argument* for a *template-parameter* T names a type that is a reference to a type A, an attempt to create the type "lvalue reference to *cv* T" creates the type "lvalue reference to A," while an attempt to create the type "rvalue reference to *cv* T" creates the type T [*Example:*

```
template <class T> class X {
  void f(const T&);
  void g(T&&);
};
X<int&> x1;                     // X<int&>::f has the parameter type int&
                                // X<int&>::g has the parameter type int&
X<const int&&> x2;              // X<const int&&>::f has the parameter type const int&
                                // X<const int&&>::g has the parameter type const int&&
```

— *end example*]

14.3.2 Template non-type arguments [temp.arg.nontype]

1 A *template-argument* for a non-type, non-template *template-parameter* shall be one of:

— an integral constant expression; or

— the name of a non-type *template-parameter*; or

— the address of an object or function with external linkage, including function templates and function *template-id*s but excluding non-static class members, expressed as & *id-expression* where the & is optional if the name refers to a function or array, or if the corresponding *template-parameter* is a reference; or

— a constant expression that evaluates to a null pointer value (4.10); or

— a constant expression that evaluates to a null member pointer value (4.11); or

— a pointer to member expressed as described in 5.3.1.

2 [*Note:* A string literal (2.13.4) does not satisfy the requirements of any of these categories and thus is not an acceptable *template-argument*. [*Example:*

```
template<class T, char* p> class X {
  X();
  X(const char* q) { /* ... */ }
};

X<int, "Studebaker"> x1;        // error: string literal as template-argument

char p[] = "Vivisectionist";
X<int,p> x2;                    // OK
```

— *end example*] — *end note*]

3 [*Note:* Addresses of array elements and names or addresses of non-static class members are not acceptable *template-argument*s. [*Example:*

```
template<int* p> class X { };

int a[10];
struct S { int m; static int s; } s;

X<&a[2]> x3;                    // error: address of array element
X<&s.m> x4;                     // error: address of non-static member
```

```
X<&s.s> x5;                     // error: &S::s must be used
X<&S::s> x6;                    // OK: address of static member
```

— *end example*] — *end note*]

4 [*Note:* Temporaries, unnamed lvalues, and named lvalues that do not have external linkage are not acceptable *template-argument*s when the corresponding *template-parameter* has reference type. [*Example:*

```
template<const int& CRI> struct B { /* ... */ };

B<1> b2;                        // error: temporary would be required for template argument

int c = 1;
B<c> b1;                        // OK
```

— *end example*] — *end note*]

5 The following conversions are performed on each expression used as a non-type *template-argument*. If a non-type *template-argument* cannot be converted to the type of the corresponding *template-parameter* then the program is ill-formed.

— for a non-type *template-parameter* of integral or enumeration type, integral promotions (4.5) and integral conversions (4.7) are applied.

— for a non-type *template-parameter* of type pointer to object, qualification conversions (4.4) and the array-to-pointer conversion (4.2) are applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (4.10) is applied. [*Note:* In particular, neither the null pointer conversion for a zero-valued integral constant expression (4.10) nor the derived-to-base conversion (4.10) are applied. Although `0` is a valid *template-argument* for a non-type *template-parameter* of integral type, it is not a valid *template-argument* for a non-type *template-parameter* of pointer type. However, both `(int*)0` and `nullptr` are valid *template-argument*s for a non-type *template-parameter* of type "pointer to int." — *end note*]

— For a non-type *template-parameter* of type reference to object, no conversions apply. The type referred to by the reference may be more cv-qualified than the (otherwise identical) type of the *template-argument*. The *template-parameter* is bound directly to the *template-argument*, which shall be an lvalue.

— For a non-type *template-parameter* of type pointer to function, the function-to-pointer conversion (4.3) is applied; if the *template-argument* is of type `std::nullptr_t`, the null pointer conversion (4.10) is applied. If the *template-argument* represents a set of overloaded functions (or a pointer to such), the matching function is selected from the set (13.4).

— For a non-type *template-parameter* of type reference to function, no conversions apply. If the *template-argument* represents a set of overloaded functions, the matching function is selected from the set (13.4).

— For a non-type *template-parameter* of type pointer to member function, if the *template-argument* is of type `std::mullptr_t`, the null member pointer conversion (4.11) is applied; otherwise, no conversions apply. If the *template-argument* represents a set of overloaded member functions, the matching member function is selected from the set (13.4).

— For a non-type *template-parameter* of type pointer to data member, qualification conversions (4.4) are applied; if the *template-argument* is of type `std::nullptr_t`, the null member pointer conversion (4.11) is applied.

[*Example:*

```
template<const int* pci> struct X { /* ... */ };
int ai[10];
X<ai> xi;                       // array to pointer and qualification conversions

struct Y { /* ... */ };
template<const Y& b> struct Z { /* ... */ };
Y y;
Z<y> z;                         // no conversion, but note extra cv-qualification

template<int (&pa)[5]> struct W { /* ... */ };
int b[5];
W<b> w;                         // no conversion

void f(char);
void f(int);

template<void (*pf)(int)> struct A { /* ... */ };

A<&f> a;                        // selects f(int)
```

— *end example*]

14.3.3 Template template arguments [temp.arg.template]

1 A *template-argument* for a template *template-parameter* shall be the name of a class template or a template alias, expressed as *id-expression*. When the *template-argument* names a class template, only primary class templates are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.

2 Any partial specializations (14.5.5) associated with the primary class template are considered when a specialization based on the template *template-parameter* is instantiated. If a specialization is not visible at the point of instantiation, and it would have been selected had it been visible, the program is ill-formed; no diagnostic is required. [*Example:*

```
template<class T> class A {     // primary template
  int x;
};
template<class T> class A<T*> { // partial specialization
  long x;
};
template<template<class U> class V> class C {
  V<int>  y;
  V<int*> z;
};
C<A> c;                         // V<int> within C<A> uses the primary template,
                                // so c.y.x has type int
                                // V<int*> within C<A> uses the partial specialization,
                                // so c.z.x has type long
```

— *end example*]

[*Example:*

```
template<class T> class A { /* ... */ };
```

```
template<class T, class U = T> class B { /* ... */ };
template <class ... Types> class C { /* ... */ };

template<template<class> class P> class X { /* ... */ };
template<template<class ...> class Q> class Y { /* ... */ };

X<A> xa;            // OK
X<B> xb;            // ill-formed: default arguments for the parameters of a template argument are ignored
X<C> xc;            // ill-formed: a template parameter pack does not match a template parameter

Y<A> ya;            // ill-formed: a template parameter pack does not match a template parameter
Y<B> yb;            // ill-formed: a template parameter pack does not match a template parameter
Y<C> yc;            // OK
```

— *end example*]

3 A *template-argument* matches a template *template-parameter* (call it P) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or template alias (call it A) matches the corresponding template parameter in the *template-parameter-list* of P. When P's *template-parameter-list* contains a template parameter pack (14.5.3), the template parameter pack will match zero or more template parameters or template parameter packs in the *template-parameter-list* of A with the same type and form as the template parameter pack in P (ignoring whether those template parameters are template parameter packs) [*Example:*

```
template <class T> struct eval;

template <template <class, class...> class TT, class T1, class... Rest>
struct eval<TT<T1, Rest...>> { };

template <class T1> struct A;
template <class T1, class T2> struct B;
template <int N> struct C;
template <class T1, int N> struct D;
template <class T1, class T2, int N = 17> struct E;

eval<A<int>> eA;                // OK: matches partial specialization of eval
eval<B<int, float>> eB;         // OK: matches partial specialization of eval
eval<C<17>> eC;                 // error: C does not match TT in partial specialization
eval<D<int, 17>> eD;            // error: D does not match TT in partial specialization
eval<E<int, float>> eE;         // error: E does not match TT in partial specialization
```

— *end example*]

14.4 Type equivalence [temp.type]

1 Two *template-id*s refer to the same class or function if

— their *template-name*s or *operator-function-id*s refer to the same template, and

— their corresponding type *template-argument*s are the same type, and

— their corresponding non-type template arguments of integral or enumeration type have identical values, and

— their corresponding non-type *template-argument*s of pointer type refer to the same external object or function or are both the null pointer value, and

— their corresponding non-type *template-argument*s of pointer-to-member type refer to the same class member or are both the null member pointer value, and

— their corresponding non-type *template-argument*s of reference type refer to the same external object or function, and

— their corresponding template *template-argument*s refer to the same template.

[*Example:*

```
template<class E, int size> class buffer { /* ... */ };
buffer<char,2*512> x;
buffer<char,1024> y;
```

declares `x` and `y` to be of the same type, and

```
template<class T, void(*err_fct)()> class list { /* ... */ };
list<int,&error_handler1> x1;
list<int,&error_handler2> x2;
list<int,&error_handler2> x3;
list<char,&error_handler2> x4;
```

declares `x2` and `x3` to be of the same type. Their type differs from the types of `x1` and `x4`.

```
template<template<class> class TT> struct X { };
template<class> struct Y { };
template<class T> using Z = Y<T>;
X<Y> y;
X<Z> z;
```

declares `y` and `z` to be of the same type. — *end example*]

2 In a constrained context (14.10), two types are the same type if some same-type requirement (14.10.1) makes them equivalent (14.10.1).

14.5 Template declarations [temp.decls]

1 A *template-id*, that is, the *template-name* followed by a *template-argument-list* shall not be specified in the declaration of a primary template declaration. [*Example:*

```
template<class T1, class T2, int I> class A<T1, T2, I> { };      // error
template<class T1, int I> void sort<T1, I>(T1 data[I]);          // error
```

— *end example*] [*Note:* however, this syntax is allowed in class template partial specializations (14.5.5). — *end note*]

2 For purposes of name lookup and instantiation, default arguments of function templates and default arguments of member functions of class templates are considered definitions; each default argument is a separate definition which is unrelated to the function template definition or to any other default arguments.

3 Because an *alias-declaration* cannot declare a *template-id*, it is not possible to partially or explicitly specialize a template alias.

14.5.1 Class templates [temp.class]

1 A class *template* defines the layout and operations for an unbounded set of related types. [*Example:* a single class template `List` might provide a common definition for list of `int`, list of `float`, and list of pointers to `Shape`s. — *end example*]

[*Example:* An array class template might be declared like this:

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};
```

2 The prefix `template <class T>` specifies that a template is being declared and that a *type-name* `T` will be used in the declaration. In other words, `Array` is a parameterized type with `T` as its parameter. — *end example*]

3 When a member function, a member class, a static data member or a member template of a class template is defined outside of the class template definition, the member definition is defined as a template definition in which the *template-parameter*s are those of the class template. The names of the template parameters used in the definition of the member may be different from the template parameter names used in the class template definition. The template argument list following the class template name in the member definition shall name the parameters in the same order as the one used in the template parameter list of the member. Each template parameter pack shall be expanded with an ellipsis in the template argument list. [*Example:*

```
template<class T1, class T2> struct A {
  void f1();
  void f2();
};

template<class T2, class T1> void A<T2,T1>::f1() { }    // OK
template<class T2, class T1> void A<T1,T2>::f2() { }    // error

template<class ... Types> struct B {
  void f3();
  void f4();
};

template<class ... Types> void B<Types ...>::f3() { }    // OK
template<class ... Types> void B<Types>::f4() { }        // error
```

— *end example*]

4 In a redeclaration, partial specialization, explicit specialization or explicit instantiation of a class template, the *class-key* shall agree in kind with the original class template declaration (7.1.6.3).

5 A constrained member (9.2) in a class template is declared only in class template specializations in which its template requirements (14.10.1) are satisfied (14.10.1.1). If there exist multiple overloads of the constrained member with identical signatures, ignoring the template requirements, and identical return types, only the most specialized overload, as determined by partial ordering of the template requirements (14.5.6.2), is declared in the instantiation. If partial ordering results in an ambiguity, a deleted function with the given signature (without any template requirements) is declared in the instantiation. [*Example:*

```
auto concept C<typename T> {
  bool operator<(T, T);
}

concept D<typename T> : C<T> { /* ... */ }
```

```
template<typename T>
struct A {
  requires C<T> void g(); // #1
  requires D<T> void g(); // #2
};

struct X { };
concept_map D<int> { /* ... */ }

void f(A<float> lf, A<int> li, A<X> lX)
{
  lf.g(); // OK: C<float> implicitly defined, calls #1
  li.g(); // OK: calls #2, which is more specialized than #1
  lX.g(); // error: no 'g' member in A<X>
}
```

— *end example*]

14.5.1.1 Member functions of class templates [temp.mem.func]

1 A member function of a class template may be defined outside of the class template definition in which it is declared. [*Example:*

```
template<class T> class Array {
  T* v;
  int sz;
public:
  explicit Array(int);
  T& operator[](int);
  T& elem(int i) { return v[i]; }
};
```

declares three function templates. The subscript function might be defined like this:

```
template<class T> T& Array<T>::operator[](int i) {
  if (i<0 || sz<=i) error("Array: range error");
  return v[i];
}
```

— *end example*]

2 The *template-argument*s for a member function of a class template are determined by the *template-argument*s of the type of the object for which the member function is called. [*Example:* the *template-argument* for `Array<T>::operator[]()` will be determined by the `Array` to which the subscripting operation is applied.

```
Array<int> v1(20);
Array<dcomplex> v2(30);

v1[3] = 7;                      // Array<int>::operator[]()
v2[3] = dcomplex(7,8);          // Array<dcomplex>::operator[]()
```

— *end example*]

14.5.1.2 Member classes of class templates [temp.mem.class]

1 A class member of a class template may be defined outside the class template definition in which it is declared. [*Note:* the class member must be defined before its first use that requires an instantiation (14.7.1). For example,

```
template<class T> struct A {
  class B;
};
A<int>::B* b1;                  // OK: requires A to be defined but not A::B
template<class T> class A<T>::B { };
A<int>::B  b2;                  // OK: requires A::B to be defined
```

— *end note*]

14.5.1.3 Static data members of class templates [temp.static]

1 A definition for a static data member may be provided in a namespace scope enclosing the definition of the static member's class template. [*Example:*

```
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;
```

— *end example*]

14.5.2 Member templates [temp.mem]

1 A template can be declared within a class or class template; such a template is called a member template. A member template can be defined within or outside its class definition or class template definition. A member template of a class template that is defined outside of its class template definition shall be specified with the *template-parameter*s of the class template followed by the *template-parameter*s of the member template. [*Example:*

```
template<class T> struct string {
  template<class T2> int compare(const T2&);
  template<class T2> string(const string<T2>& s) { /* ... */ }
};

template<class T> template<class T2> int string<T>::compare(const T2& s) {
}
```

— *end example*]

2 A local class shall not have member templates. Access control rules (Clause 11) apply to member template names. A destructor shall not be a member template. A normal (non-template) member function with a given name and type and a member function template of the same name, which could be used to generate a specialization of the same type, can both be declared in a class. When both exist, a use of that name and type refers to the non-template member unless an explicit template argument list is supplied. [*Example:*

```
template <class T> struct A {
  void f(int);
  template <class T2> void f(T2);
};
```

```
template <> void A<int>::f(int) { }                    // non-template member
template <> template <> void A<int>::f<>(int) { }      // template member

int main() {
  A<char> ac;
  ac.f(1);          // non-template
  ac.f('c');        // template
  ac.f<>(1);        // template
}
```

— *end example*]

3 A member function template shall not be virtual. [*Example:*

```
template <class T> struct AA {
  template <class C> virtual void g(C);   // error
  virtual void f();                       // OK
};
```

— *end example*]

4 A specialization of a member function template does not override a virtual function from a base class. [*Example:*

```
class B {
  virtual void f(int);
};

class D : public B {
  template <class T> void f(T); // does not override B::f(int)
  void f(int i) { f<>(i); }     // overriding function that calls
                                // the template instantiation
};
```

— *end example*]

5 A specialization of a conversion function template is referenced in the same way as a non-template conversion function that converts to the same type. [*Example:*

```
struct A {
  template <class T> operator T*();
};
template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; }    // specialization
template A::operator void*();                   // explicit instantiation

int main() {
  A a;
  int *ip;
  ip = a.operator int*();       // explicit call to template operator
                                // A::operator int*()
}
```

— *end example*] [*Note:* because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called

without using a function name, there is no way to provide an explicit template argument list for these function templates. — *end note*]

6 A specialization of a conversion function template is not found by name lookup. Instead, any conversion function templates visible in the context of the use are considered. For each such operator, if argument deduction succeeds (14.8.2.3), the resulting specialization is used as if found by name lookup.

7 A *using-declaration* in a derived class cannot refer to a specialization of a conversion function template in a base class.

8 Overload resolution (13.3.3.2) and partial ordering (14.5.6.2) are used to select the best conversion function among multiple specializations of conversion function templates and/or non-template conversion functions.

9 A member template of a constrained class template is itself a constrained template (14.10). The template requirements of the member template are the template requirements of each of its enclosing constrained templates and any requirements specified or implied by the member template itself. [*Example:*

```
concept C<typename T> { void f(const T&); }
concept D<typename T> { void g(const T&); }

template<C T> class A {
  requires D<T> void h(const T& x) {
    f(x); // OK: C<T>::f
    g(x); // OK: D<T>::g
  }
};
```

— *end example*]

14.5.3 Variadic templates [temp.variadic]

1 A *template parameter pack* is a template parameter that accepts zero or more template arguments. [*Example:*

```
template<class ... Types> struct Tuple { };

Tuple<> t0;                    // Types contains no arguments
Tuple<int> t1;                 // Types contains one argument: int
Tuple<int, float> t2;          // Types contains two arguments: int and float
Tuple<0> eror;                 // error: 0 is not a type
```

— *end example*]

[*Note:* a template parameter pack can also occur in a concept's template parameter list (14.9.1). [*Example:*

```
auto concept C<typename F, typename... Args> {
  typename result_type;
  result_type operator()(F&, Args...);
}
```

— *end example*] — *end note*]

2 A *function parameter pack* is a function parameter that accepts zero or more function arguments. [*Example:*

```
template<class ... Types> void f(Types ... args);

f();               // OK: args contains no arguments
f(1);              // OK: args contains one argument: int
f(2, 1.0);         // OK: args contains two arguments: int and double
```

— *end example*]

3 A *parameter pack* is either a template parameter pack or a function parameter pack.

4 A *pack expansion* is a sequence of tokens that names one or more parameter packs, followed by an ellipsis. The sequence of tokens is called the *pattern of the expansion*; its syntax depends on the context in which the expansion occurs. Pack expansions can occur in the following contexts:

— In an *expression-list* (5.2); the pattern is an *initializer-clause*.

— In an *initializer-list* (8.5); the pattern is an *initializer-clause*.

— In a *base-specifier-list* (10); the pattern is a *base-specifier*.

— In a *mem-initializer-list* (12.6.2); the pattern is a *mem-initializer*.

— In a *template-argument-list* (14.3); the pattern is a *template-argument*.

— In an *exception-specification* (15.4); the pattern is a *type-id*.

— In a *requirement-list* (14.10.1); the pattern is a *requirement*.

[*Example:*

```
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
  f(&rest ...);      // "&rest ..." is a pack expansion; "&rest" is its pattern
}
```

— *end example*]

5 A parameter pack whose name appears within the pattern of a pack expansion is expanded by that pack expansion. An appearance of the name of a parameter pack is only expanded by the innermost enclosing pack expansion. The pattern of a pack expansion shall name one or more parameter packs that are not expanded by a nested pack expansion. All of the parameter packs expanded by a pack expansion shall have the same number of arguments specified. An appearance of a name of a parameter pack that is not expanded is ill-formed. [*Example:*

```
template<typename...> struct Tuple {};
template<typename T1, typename T2> struct Pair {};

template<class ... Args1> struct zip {
  template<class ... Args2> struct with {
    typedef Tuple<Pair<Args1, Args2> ... > type;
  };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
    // T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>
typedef zip<short>::with<unsigned short, unsigned>::type T2;
    // error: different number of arguments specified for Args1 and Args2

template<class ... Args> void g(Args ... args) {
  f(const_cast<const Args*>(&args)...); // OK: "Args" and "args" are expanded
  f(5 ...);                             // error: pattern does not contain any parameter packs
  f(args);                              // error: parameter pack "args" is not expanded
  f(h(args ...) + args ...);            // OK: first "args" expanded within h, second "args" expanded
within f
  }
```

— *end example*]

6 The instantiation of an expansion produces a list $\texttt{E}_1 \oplus \texttt{E}_2 \oplus ... \oplus \texttt{E}_N$, where N is the number of elements in the pack expansion parameters and $\oplus$ is the syntactically-appropriate separator for the list. Each $\texttt{E}_i$ is generated by instantiating the pattern and replacing each pack expansion parameter with its ith element. All of the $\texttt{E}_i$ become elements in the enclosing list. [*Note:* The variety of list varies with the context: *expression-list*, *base-specifier-list*, *template-argument-list*, *requirement-list*, etc. — *end note*]

14.5.4 Friends [temp.friend]

1 A friend of a class or class template can be a function template or class template, a specialization of a function template or class template, or an ordinary (non-template) function or class. For a friend function declaration that is not a template declaration:

— if the name of the friend is a qualified or unqualified *template-id*, the friend declaration refers to a specialization of a function template, otherwise

— if the name of the friend is a *qualified-id* and a matching non-template function is found in the specified class or namespace, the friend declaration refers to that function, otherwise,

— if the name of the friend is a *qualified-id* and a matching specialization of a function template is found in the specified class or namespace, the friend declaration refers to that function template specialization, otherwise,

— the name shall be an *unqualified-id* that declares (or redeclares) an ordinary (non-template) function.

[*Example:*

```
template<class T> class task;
template<class T> task<T>* preempt(task<T>*);

template<class T> class task {
  friend void next_time();
  friend void process(task<T>*);
  friend task<T>* preempt<T>(task<T>*);
  template<class C> friend int func(C);

  friend class task<int>;
  template<class P> friend class frd;
};
```

Here, each specialization of the `task` class template has the function `next_time` as a friend; because `process` does not have explicit *template-argument*s, each specialization of the `task` class template has an appropriately typed function `process` as a friend, and this friend is not a function template specialization; because the friend `preempt` has an explicit *template-argument* `<T>`, each specialization of the `task` class template has the appropriate specialization of the function template `preempt` as a friend; and each specialization of the `task` class template has all specializations of the function template `func` as friends. Similarly, each specialization of the `task` class template has the class template specialization `task<int>` as a friend, and has all specializations of the class template `frd` as friends. — *end example*]

2 A friend template may be declared within a class or class template. A friend function template may be defined within a class or class template, but a friend class template may not be defined in a class or class template. In these cases, all specializations of the friend class or friend function template are friends of the class or class template granting friendship. [*Example:*

```
class A {
  template<class T> friend class B;                 // OK
  template<class T> friend void f(T){ /* ...  */ }  // OK
};
```

— *end example*]

3 A template friend declaration specifies that all specializations of that template, whether they are implicitly instantiated (14.7.1), partially specialized (14.5.5) or explicitly specialized (14.7.3), are friends of the class containing the template friend declaration. [*Example:*

```
class X {
  template<class T> friend struct A;
  class Y { };
};

template<class T> struct A { X::Y ab; };            // OK
template<class T> struct A<T*> { X::Y ab; };        // OK
```

— *end example*]

4 When a function is defined in a friend function declaration in a class template, the function is instantiated when the function is used. The same restrictions on multiple declarations and definitions that apply to non-template function declarations and definitions also apply to these implicit definitions.

5 A member of a class template may be declared to be a friend of a non-template class. In this case, the corresponding member of every specialization of the class template is a friend of the class granting friendship. [*Example:*

```
template<class T> struct A {
  struct B { };
  void f();
};

class C {
  template<class T> friend struct A<T>::B;
  template<class T> friend void A<T>::f();
};
```

— *end example*]

6 [*Note:* a friend declaration may first declare a member of an enclosing namespace scope (14.6.5). — *end note*]

7 A friend template shall not be declared in a local class.

8 Friend declarations shall not declare partial specializations. [*Example:*

```
template<class T> class A { };
class X {
  template<class T> friend class A<T*>; // error
};
```

— *end example*]

9 When a friend declaration refers to a specialization of a function template, the function parameter declarations shall not include default arguments, nor shall the inline specifier be used in such a declaration.

14.5.5 Class template partial specializations [temp.class.spec]

1 A *primary* class template declaration is one in which the class template name is an identifier. A template declaration in which the class template name is a *simple-template-id* is a *partial specialization* of the class template named in the *simple-template-id*. A partial specialization of a class template provides an alternative definition of the template that is used instead of the primary definition when the arguments in a specialization match those given in the partial specialization (14.5.5.1). The primary template shall be declared before any specializations of that template. A partial specialization shall be declared before the first use of a class template specialization that would make use of the partial specialization as the result of an implicit or explicit instantiation in every translation unit in which such a use occurs; no diagnostic is required.

2 When a partial specialization is used within the instantiation of an exported template, and the unspecialized template name is non-dependent in the exported template, a declaration of the partial specialization shall be declared before the definition of the exported template, in the translation unit containing that definition. A similar restriction applies to explicit specialization; see 14.7.

3 Each class template partial specialization is a distinct template and definitions shall be provided for the members of a template partial specialization (14.5.5.3).

4 [*Example:*

```
template<class T1, class T2, int I> class A             { };    // #1
template<class T, int I>            class A<T, T*, I>   { };    // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { };    // #3
template<class T>                   class A<int, T*, 5> { };    // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { };    // #5
```

The first declaration declares the primary (unspecialized) class template. The second and subsequent declarations declare partial specializations of the primary template. — *end example*]

5 The template parameters are specified in the angle bracket enclosed list that immediately follows the keyword `template`. For partial specializations, the template argument list is explicitly written immediately following the class template name. For primary templates, this list is implicitly described by the template parameter list. Specifically, the order of the template arguments is the sequence in which they appear in the template parameter list. [*Example:* the template argument list for the primary template in the example above is `<T1, T2, I>`. — *end example*] [*Note:* the template argument list shall not be specified in the primary template declaration. For example,

```
template<class T1, class T2, int I> class A<T1, T2, I>  { };    // error
```

— *end note*]

6 A class template partial specialization may be declared or redeclared in any namespace scope in which its definition may be defined (14.5.1 and 14.5.2). [*Example:*

```
template<class T> struct A {
  struct C {
    template<class T2> struct B { };
  };
};

// partial specialization of A<T>::C::B<T2>
template<class T> template<class T2>
  struct A<T>::C::B<T2*> { };

A<short>::C::B<int*> absip;      // uses partial specialization
```

— *end example*]

7 Partial specialization declarations themselves are not found by name lookup. Rather, when the primary template name is used, any previously-declared partial specializations of the primary template are also considered. One consequence is that a *using-declaration* which refers to a class template does not restrict the set of partial specializations which may be found through the *using-declaration*. [*Example:*

```
namespace N {
  template<class T1, class T2> class A { };          // primary template
}

using N::A;                                  // refers to the primary template

namespace N {
  template<class T> class A<T, T*> { }; // partial specialization
}

A<int,int*> a;                    // uses the partial specialization, which is found through
                                  // the using declaration which refers to the primary template
```

— *end example*]

8 A non-type argument is non-specialized if it is the name of a non-type parameter. All other non-type arguments are specialized.

9 Within the argument list of a class template partial specialization, the following restrictions apply:

— A partially specialized non-type argument expression shall not involve a template parameter of the partial specialization except when the argument expression is a simple *identifier*. [*Example:*

```
template <int I, int J> struct A {};
template <int I> struct A<I+5, I*2> {}; // error

template <int I, int J> struct B {};
template <int I> struct B<I, I> {};     // OK
```

— *end example*]

— The type of a template parameter corresponding to a specialized non-type argument shall not be dependent on a parameter of the specialization. [*Example:*

```
template <class T, T t> struct C {};
template <class T> struct C<T, 1>;                  // error

template< int X, int (*array_ptr)[X] > class A {};
int array[5];
template< int X > class A<X,&array> { };            // error
```

— *end example*]

— The argument list of the specialization shall not be identical to the implicit argument list of the primary template unless the specialization contains template requirements that are more specific (14.5.6.2) than the primary template's requirements. [*Example:*

```
concept C<typename T> { int F(T); }

template<typename T> class X { /* ... */ };                    // #6
template<typename T> requires C<T> class X<T> { /* ... */ };   // #7, OK
```

— *end example*]

— The template parameter list of a specialization shall not contain default template argument values.[132]

— An argument shall not contain an unexpanded parameter pack. If an argument is a pack expansion (14.5.3), it shall be the last argument in the template argument list.

10 The template requirements of a primary class template are implied (14.10.1.2) in its class template partial specializations that are constrained templates. [*Example:*

```
concept C<typename T> { /* ... */ }
concept D<typename T> { /* ... */ }

template<typename T> requires C<T> class Y { /* ... */ };
template<typename T>
  requires D<T> // same as requires C<T> && D<T>
  class Y<T> { /* ... */ };
```

— *end example*]

14.5.5.1 Matching of class template partial specializations [temp.class.spec.match]

1 When a class template is used in a context that requires an instantiation of the class, it is necessary to determine whether the instantiation is to be generated using the primary template or one of the partial specializations. This is done by matching the template arguments of the class template specialization with the template argument lists of the partial specializations.

— If exactly one matching specialization is found, the instantiation is generated from that specialization.

— If more than one matching specialization is found, the partial order rules (14.5.5.2) are used to determine whether one of the specializations is more specialized than the others. If none of the specializations is more specialized than all of the other matching specializations, then the use of the class template is ambiguous and the program is ill-formed.

— If no matches are found, the instantiation is generated from the primary template.

2 A partial specialization matches a given actual template argument list if the template arguments of the partial specialization can be deduced from the actual template argument list (14.8.2) and the deduced template arguments satisfy the partial specializaton's template requirements (if any). [*Example:*

```
A<int, int, 1>   a1;           // uses #1
A<int, int*, 1>  a2;           // uses #2, T is int, I is 1
A<int, char*, 5> a3;           // uses #4, T is char
A<int, char*, 1> a4;           // uses #5, T1 is int, T2 is char, I is 1
A<int*, int*, 2> a5;           // ambiguous: matches #3 and #5

concept_map D<int> { /* ... */}
struct Y { }

X<int> x1;                     // uses #7
X<Y> x2;                       // uses #6
```

— *end example*]

3 A non-type template argument can also be deduced from the value of an actual template argument of a non-type parameter of the primary template. [*Example:* the declaration of `a2` above. — *end example*]

132) There is no way in which they could be used.

4 In a type name that refers to a class template specialization, (e.g., `A<int, int, 1>`) the argument list shall match the template parameter list of the primary template. If the primary template has template requirements, the arguments shall satisfy those requirements. The template arguments of a specialization are deduced from the arguments of the primary template.

14.5.5.2 Partial ordering of class template specializations [temp.class.order]

1 For two class template partial specializations, the first is at least as specialized as the second if, given the following rewrite to two function templates, the first function template is at least as specialized as the second according to the ordering rules for function templates (14.5.6.2):

— the first function template has the same template parameters as the first partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the first partial specialization, and

— the second function template has the same template parameters as the second partial specialization and has a single function parameter whose type is a class template specialization with the template arguments of the second partial specialization.

2 [*Example:*

```
concept Con1<typename T> { }
concept Con2<typename T> : Con1<T> { }
template<int I, int J, class T> class X { };
template<int I, int J>          class X<I, J, int> { }; // #1
template<int I>                 class X<I, I, int> { }; // #2
template<int I, int J, class T>
  requires Con1<T> class X<I, J, T> { };                // #3
template<int I, int J, class T>
  requires Con1<2> class X<I, J, T> { };                // #4

template<int I, int J> void f(X<I, J, int>);            // A
template<int I>        void f(X<I, I, int>);            // B
template<int I, int J, class T>
  requires Con1<T> void f(X<I, J, T>);                  // C
template<int I, int J, class T>
  requires Con2<T> void f(X<I, J, T>);                  // D
```

The partial specialization `#2` is more specialized than the partial specialization `#1` because the function template `B` is more specialized than the function template `A` according to the ordering rules for function templates. The partial specialization `#4` is more specialized than the partial specialization `#3` because the function template `D` is more specialized than the function template `C` according to the partial ordering rules for function templates. — *end example*]

14.5.5.3 Members of class template specializations [temp.class.spec.mfunc]

1 The template parameter list of a member of a class template partial specialization shall match the template parameter list of the class template partial specialization. The template argument list of a member of a class template partial specialization shall match the template argument list of the class template partial specialization. A class template specialization is a distinct template. The members of the class template partial specialization are unrelated to the members of the primary template. Class template partial specialization members that are used in a way that requires a definition shall be defined; the definitions of members of the primary template are never used as definitions for members of a class template partial specialization. An

explicit specialization of a member of a class template partial specialization is declared in the same way as an explicit specialization of the primary template. [*Example:*

```
// primary template
template<class T, int I> struct A {
  void f();
};

template<class T, int I> void A<T,I>::f() { }

// class template partial specialization
template<class T> struct A<T,2> {
  void f();
  void g();
  void h();
};

// member of class template partial specialization
template<class T> void A<T,2>::g() { }

// explicit specialization
template<> void A<char,2>::h() { }

int main() {
  A<char,0> a0;
  A<char,2> a2;
  a0.f();                       // OK, uses definition of primary template's member
  a2.g();                       // OK, uses definition of
                                // partial specialization's member
  a2.h();                       // OK, uses definition of
                                // explicit specialization's member
  a2.f();                       // ill-formed, no definition of f for A<T,2>
                                // the primary template is not used here
}
```

— *end example*]

2 If a member template of a class template is partially specialized, the member template partial specializations are member templates of the enclosing class template; if the enclosing class template is instantiated (14.7.1, 14.7.2), a declaration for every member template partial specialization is also instantiated as part of creating the members of the class template specialization. If the primary member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the partial specializations of the member template are ignored for this specialization of the enclosing class template. If a partial specialization of the member template is explicitly specialized for a given (implicit) specialization of the enclosing class template, the primary member template and its other partial specializations are still considered for this specialization of the enclosing class template. [*Example:*

```
template<class T> struct A {
  template<class T2> struct B {};                   // #1
  template<class T2> struct B<T2*> {};              // #2
};

template<> template<class T2> struct A<short>::B {};  // #3

A<char>::B<int*>  abcip;  // uses #2
A<short>::B<int*> absip;  // uses #3
```

```
A<char>::B<int>  abci;     // uses #1
```

— *end example*]

14.5.6 Function templates [temp.fct]

1 A function template defines an unbounded set of related functions. [*Example:* a family of sort functions might be declared like this:

```
template<class T> class Array { };
template<class T> void sort(Array<T>&);
```

— *end example*]

2 A function template can be overloaded with other function templates and with normal (non-template) functions. A normal function is not related to a function template (i.e., it is never considered to be a specialization), even if it has the same name and type as a potentially generated function template specialization.[133]

14.5.6.1 Function template overloading [temp.over.link]

1 It is possible to overload function templates so that two different function template specializations have the same type. [*Example:*

```
// file1.c
template<class T>
  void f(T*);
void g(int* p) {
  f(p); // calls f<int>(int*)
}
```

```
// file2.c
template<class T>
  void f(T);
void h(int* p) {
  f(p); // calls f<int*>(int*)
}
```

— *end example*]

2 Such specializations are distinct functions and do not violate the one definition rule (3.2).

3 The signature of a function template is defined in 1.3. The names of the template parameters are significant only for establishing the relationship between the template parameters and the rest of the signature. [*Note:* two distinct function templates may have identical function return types and function parameter lists, even if overload resolution alone cannot distinguish them.

```
template<class T> void f();
template<int I> void f();        // OK: overloads the first template
                                 // distinguishable with an explicit template argument list
```

— *end note*]

4 When an expression that references a template parameter is used in the function parameter list or the return type in the declaration of a function template, the expression that references the template parameter is part of the signature of the function template. This is necessary to permit a declaration of a function template in one translation unit to be linked with another declaration of the function template in another translation unit and, conversely, to ensure that function templates that are intended to be distinct are not linked with one another. [*Example:*

133) That is, declarations of non-template functions do not merely guide overload resolution of function template specializations with the same name. If such a non-template function is used in a program, it must be defined; it will not be implicitly instantiated using the function template definition.

```
template <int I, int J> A<I+J> f(A<I>, A<J>);   // #1
template <int K, int L> A<K+L> f(A<K>, A<L>);   // same as #1
template <int I, int J> A<I-J> f(A<I>, A<J>);   // different from #1
```

— *end example*] [*Note:* Most expressions that use template parameters use non-type template parameters, but it is possible for an expression to reference a type parameter. For example, a template type parameter can be used in the `sizeof` operator. — *end note*]

5 Two expressions involving template parameters are considered *equivalent* if two function definitions containing the expressions would satisfy the one definition rule (3.2), except that the tokens used to name the template parameters may differ as long as a token used to name a template parameter in one expression is replaced by another token that names the same template parameter in the other expression. [*Example:*

```
template <int I, int J> void f(A<I+J>);         // #1
template <int K, int L> void f(A<K+L>);         // same as #1
```

— *end example*] Two expressions involving template parameters that are not equivalent are *functionally equivalent* if, for any given set of template arguments, the evaluation of the expression results in the same value.

6 Two function templates are *equivalent* if they are declared in the same scope, have the same name, have identical template parameter lists, have identical template requirements (if any), and have return types and parameter lists that are equivalent using the rules described above to compare expressions involving template parameters. Two function templates are *functionally equivalent* if they are equivalent except that one or more expressions that involve template parameters in the return types, parameter lists, and template requirements (if any) are functionally equivalent using the rules described above to compare expressions involving template parameters. If a program contains declarations of function templates that are functionally equivalent but not equivalent, the program is ill-formed; no diagnostic is required.

7 [*Note:* This rule guarantees that equivalent declarations will be linked with one another, while not requiring implementations to use heroic efforts to guarantee that functionally equivalent declarations will be treated as distinct. For example, the last two declarations are functionally equivalent and would cause a program to be ill-formed:

```
// Guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+10>);

// Guaranteed to be different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// Ill-formed, no diagnostic required
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
```

— *end note*]

14.5.6.2 Partial ordering of function templates [temp.func.order]

1 If a function template is overloaded, the use of a function template specialization might be ambiguous because template argument deduction (14.8.2) may associate the function template specialization with more than one function template declaration. *Partial ordering* of overloaded function template declarations is used in the following contexts to select the function template to which a function template specialization refers:

— during overload resolution for a call to a function template specialization (13.3.3);

— when the address of a function template specialization is taken;

— when a placement operator delete that is a function template specialization is selected to match a placement operator new (3.7.4.2, 5.3.4);

— when a friend function declaration (14.5.4), an explicit instantiation (14.7.2) or an explicit specialization (14.7.3) refers to a function template specialization.

2 Partial ordering selects which of two function templates is more specialized than the other by transforming each template in turn (see next paragraph) and performing template argument deduction using the function parameter types, or in the case of a conversion function the return type. [*Note:* if template argument deduction succeeds, then the template requirements (if any) have all been satisfied (14.10.1.1) by the deduced template arguments. — *end note*] The deduction process determines whether one of the templates is more specialized than the other. If so, the more specialized template is the one chosen by the partial ordering process.

3 To produce the transformed template, for each type, non-type, or template template parameter (including template parameter packs thereof) synthesize a unique type, value, or class template respectively and substitute it for each occurrence of that parameter in the function type of the template. When the template is a constrained template, the unique type is an archetype and concept map archetypes for each of the requirements stated in or implied by its template requirements are also synthesized; see 14.10. [*Note:* because the unique types are archetypes, two template type parameters may share the same archetype due to same-type constraints. — *end note*]

4 Using the transformed function template's function parameter list, or in the case of a conversion function its transformed return type, perform type deduction against the function parameter list (or return type) of the other function. The mechanism for performing these deductions is given in 14.8.2.4.

[*Example:*

```
template<class T> struct A { A(); };

template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);

template<class T> void g(T);
template<class T> void g(T&);

template<class T> void h(const T&);
template<class T> void h(A<T>&);

void m() {
  const int *p;
  f(p);              // f(const T*) is more specialized than f(T) or f(T*)
  float x;
  g(x);              // Ambiguous: g(T) or g(T&)
  A<int> z;
  h(z);              // overload resolution selects h(A<T>&)
  const A<int> z2;
  h(z2);             // h(const T&) is called because h(A<T>&) is not callable
}
```

— *end example*]

[*Note:* when two constrained templates have identical signatures (ignoring template requirements), the partial ordering is based on those template requirements. Similarly, a constrained template is more specialized than an unconstrained template because it has more stringent requirements. — *end note*] [*Example:*

```
auto concept CopyConstructible<typename T> {
  T::T(const T&);
}

template<CopyConstructible T> struct A { A(); };

concept C<typename T> { }
concept D<typename T> : C<T> { }
concept_map C<int*> { }
concept_map D<float> { }
template<typename T> concept_map D<A<T>> { }

template<class T> requires C<T> void f(T&) { } // #1
template<class T> requires D<T> void f(T&) { } // #2
template<class T> requires C<A<T>> void f(A<T>&) { } // #3
template<class T> void f(T&); // #4

void m() {
  int *p;
  f(p);          // calls #1: template argument deductions fails #2 and #3, and #1 is more specialized than
#4
  float x;
  f(x);          // #2 is called because #3 is not callable and #2 is more specialized than #1 and #4
  A<int> z;
  f(z);          // ambiguous: no partial ordering between #2 and #3
}
```

— *end example*]

5 The presence of unused ellipsis and default arguments has no effect on the partial ordering of function templates. [*Example:*

```
template<class T> void f(T);           // #1
template<class T> void f(T*, int=1);   // #2
template<class T> void g(T);           // #3
template<class T> void g(T*, ...);     // #4

int main() {
  int* ip;
  f(ip);           // calls #2
  g(ip);           // calls #4
}
```

— *end example*]

14.5.7 Template aliases [temp.alias]

1 A *template alias* declares a name for a family of types. The name of the template alias is a *template-name*.

2 When a *template-id* refers to the specialization of a template alias, it is equivalent to the associated type obtained by substitution of its *template-argument*s for the *template-parameter*s in the *type-id* of the template alias. [*Note:* A template alias name is never deduced. — *end note*] [*Example:*

```
template<class T> struct Alloc { /* ... */ };
template<class T> using Vec = vector<T, Alloc<T>>;
Vec<int> v;          // same as vector<int, Alloc<int>> v;

template<class T>
  void process(Vec<T>& v)
  { /* ... */ }

template<class T>
  void process(vector<T, Alloc<T>>& w)
  { /* ... */ }      // error: redefinition

template<template<class> class TT>
  void f(TT<int>);

f(v);                // error: Vec not deduced

template<template<class,class> class TT>
  void g(TT<int, Alloc<int>>);
g(v);                // OK: TT = vector
```

— *end example*]

14.5.8 Concept map templates [temp.concept.map]

1 A *concept map template* defines an unbounded set of concept maps with a common set of associated function, associated type, and associated class template definitions. [*Example:*

```
concept F<typename T> {
  typename type;
  type f(T);
}

template<typename T>
concept_map F<T*> {
  typedef T& type;
  T& f(T*);
}
```

— *end example*]

2 A concept map template is a constrained template (14.10) [*Note:* a concept map template is a constrained template even if it does not have template requirements. — *end note*]

3 Within the *template-argument-list* of the *concept-id* in a concept map template (including nested template argument lists), the following restrictions apply:

— A non-type argument expression shall not involve a template parameter of the concept map except when the argument expression is a simple *identifier*.

— The type of a template parameter corresponding to a non-type argument shall not be dependent on a parameter of the concept map.

— The template parameter list of a concept map template shall not contain default template argument values.[134]

134) There is no way in which they could be used.

4 During concept map lookup (14.10.1.1), concept map matching determines whether a particular concept map template can be used. Concept map matching matches the template arguments in the *concept instance* to the template arguments in the concept map template, using matching of class template partial specializations (14.5.5.1).

5 For two concept map templates, the first is at least as specialized as the second if, given the following rewrite to two class template partial specializations of an invented class template X, the first class template partial specialization is at least as specialized as the second according to the rules for partial ordering of class template partial specializations (14.5.5.2). The primary class template X has the same template parameters as the concept of the concept map templates. The class template partial specializations are constrained templates, even if the corresponding concept map templates have no requirements specified.

— the first class template partial specialization has the same template parameters and template arguments as the first concept map template, and

— the second class template has the same template parameters and template arguments as the second concept map template.

[*Example:*

```
concept C<typename T> { }
concept Ptr<typename T> { }
template<typename T> concept_map Ptr<T*> { /* ... */ }

template<typename T> requires Ptr<T*> concept_map C<T*> { /* ... */ } // #1
template<typename T> requires Ptr<T> concept_map C<T> { /* ... */ }   // #2

template<typename T> class X;
template<typename T> requires Ptr<T*> class X<T*>;                    // A
template<typename T> requires Ptr<T> class X<T>;                      // B
```

The concept map template #1 is more specialized than the concept map template #2 because the class template partial specialization A is more specialized than the class template partial specialization B according to the ordering rules for class template partial specializations. — *end example*]

6 A concept map template shall satisfy the requirements of its corresponding concept (14.9.2) at the time of definition of the concept map template. [*Example:*

```
concept C<typename T> { }

concept F<typename T> {
  void f(T);
}

template<C T> struct X;

template<F T> void f(X<T>);      // #1

template<typename T>
concept_map F<X<T>> { }          // error: requirement for f(X<T>) not satisfied

template<F T>
concept_map F<X<T>> { }          // OK: uses #1 to satisfy requirement for f(X<T>)
```

— *end example*]

7 If the definition of a concept map template uses an instantiated archetype (14.10.2), and instantiation of the concept map template results in a different specialization of that class template with an incompatible definition, the program is ill-formed. The specialization is considered to have an incompatible definition if the specialization's definition causes a different definition of any associated type or associated class template in the concept map, if its definition causes any of the associated function definitions to be ill-formed, or if the resulting concept map fails to satisfy the axioms of the corresponding concept. [*Example:*

```
concept Stack<typename X> {
  typename value_type;
  value_type& top(X&);
  // ...
}

template<typename T> struct dynarray {
  T& top();
};

template<> struct dynarray<bool> {
  bool top();
};

template<typename T>
concept_map Stack<dynarray<T>> {
  typedef T value_type;
  T& top(dynarray<T>& x) { return x.top(); }
}

template<Stack X>
void f(X& x) {
  X::value_type& t = top(x);
}

void g(dynarray<int>& x1, dynarray<bool>& x2) {
  f(x1); // OK
  f(x2); // error: Stack<dynarray<bool> > uses the dynarray<bool> class specialization
         // rather than the dynarray primary class template, and the two
         // have incompatible signatures for top()
}
```

— *end example*]

8 A concept map template shall be declared before the first use of a concept map that would make use of the concept map template as the result of an instantiation in every translation unit in which such a use occurs; no diagnostic is required.

14.6 Name resolution [temp.res]

1 Three kinds of names can be used within a template definition:

— The name of the template itself, and names declared within the template itself.

— Names dependent on a *template-parameter* (14.6.2).

— Names from scopes which are visible within the template definition.

2 A name used in a template declaration or definition and that is dependent on a *template-parameter* is assumed not to name a type unless the applicable name lookup finds a type name or the name is qualified by the keyword `typename`. [*Example:*

```
// no B declared here

class X;

template<class T> class Y {
  class Z;                        // forward declaration of member class

  void f() {
    X* a1;                        // declare pointer to X
    T* a2;                        // declare pointer to T
    Y* a3;                        // declare pointer to Y<T>
    Z* a4;                        // declare pointer to Z
    typedef typename T::A TA;
    TA* a5;                       // declare pointer to T's A
    typename T::A* a6;            // declare pointer to T's A
    T::A* a7;                     // T::A is not a type name:
                                  // multiply T::A by a7; ill-formed,
                                  // no visible declaration of a7
    B* a8;                        // B is not a type name:
                                  // multiply B by a8; ill-formed,
                                  // no visible declarations of B and a8
  }
};
```

— *end example*]

3 When a *qualified-id* is intended to refer to a type that is not a member of the current instantiation (14.6.2.1) and its *nested-name-specifier* is not a concept instance (14.9) and depends on a *template-parameter* (14.6.2), it shall be prefixed by the keyword `typename`, forming a *typename-specifier*. If the *qualified-id* in a *typename-specifier* does not denote a type, the program is ill-formed. When the *nested-name-specifier* refers to a concept instance, name lookup into the corresponding concept determines whether the *qualified-id* refers to a type or a value.

typename-specifier:
`typename` ::$_{opt}$ *nested-name-specifier identifier*
`typename` ::$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *simple-template-id*

4 If a specialization of a template is instantiated for a set of *template-argument*s such that the *qualified-id* prefixed by `typename` does not denote a type, the specialization is ill-formed. The usual qualified name lookup (3.4.3) is used to find the *qualified-id* even in the presence of `typename`. [*Example:*

```
struct A {
  struct X { };
  int X;
};
struct B {
  struct X { };
};
template<class T> void f(T t) {
  typename T::X x;
}
void foo() {
  A a;
```

```
  B b;
  f(b);             // OK: T::X refers to B::X
  f(a);             // error: T::X refers to the data member A::X not the struct A::X
}
```

— *end example*]

5 A qualified name used as the name in a *mem-initializer-id*, a *base-specifier*, or an *elaborated-type-specifier* is implicitly assumed to name a type, without the use of the `typename` keyword. [*Note:* the `typename` keyword is not permitted by the syntax of these constructs. — *end note*]

6 If, for a given set of template arguments, a specialization of a template is instantiated that refers to a *qualified-id* that denotes a type, and the *nested-name-specifier* of the *qualified-id* depends on a template parameter, the *qualified-id* shall either be prefixed by `typename` or shall be used in a context in which it implicitly names a type as described above. [*Example:*

```
template <class T> void f(int i) {
  T::x * i;         // T::x must not be a type
}

struct Foo {
  typedef int x;
};

struct Bar {
  static int const x = 5;
};

int main() {
  f<Bar>(1);        // OK
  f<Foo>(1);        // error: Foo::x is a type
}
```

— *end example*]

7 Within the definition of a class template or within the definition of a member of a class template, the keyword `typename` is not required when referring to the unqualified name of a previously declared member of the class template that declares a type. [*Example:*

```
template<class T> struct A {
  typedef int B;
  B b;              // OK, no typename required
};
```

— *end example*]

8 Knowing which names are type names allows the syntax of every template definition to be checked. No diagnostic shall be issued for a template definition for which a valid specialization can be generated. If no valid specialization can be generated for a template definition, and that template is not instantiated, the template definition is ill-formed, no diagnostic required. If a type used in a non-dependent name is incomplete at the point at which a template is defined but is complete at the point at which an instantiation is done, and if the completeness of that type affects whether or not the program is well-formed or affects the semantics of the program, the program is ill-formed; no diagnostic is required. [*Note:* if a template is instantiated, errors will be diagnosed according to the other rules in this Standard. Exactly when these errors are diagnosed is a quality of implementation issue. — *end note*] [*Example:*

```
int j;
template<class T> class X {
  void f(T t, int i, char* p) {
    t = i;          // diagnosed if X::f is instantiated
                    // and the assignment to t is an error
    p = i;          // may be diagnosed even if X::f is
                    // not instantiated
    p = j;          // may be diagnosed even if X::f is
                    // not instantiated
  }
  void g(T t) {
    +;              // may be diagnosed even if X::g is
                    // not instantiated
  }
};
```

— *end example*]

9 When looking for the declaration of a name used in a template definition, the usual lookup rules (3.4.1, 3.4.2) are used for non-dependent names. The lookup of names dependent on the template parameters is postponed until the actual template argument is known (14.6.2). [*Example:*

```
#include <iostream>
using namespace std;

template<class T> class Set {
  T* p;
  int cnt;
public:
  Set();
  Set<T>(const Set<T>&);
  void printall() {
    for (int i = 0; i<cnt; i++)
      cout << p[i] << '\n';
  }
};
```

in the example, `i` is the local variable `i` declared in `printall`, `cnt` is the member `cnt` declared in `Set`, and `cout` is the standard output stream declared in `iostream`. However, not every declaration can be found this way; the resolution of some names must be postponed until the actual *template-argument*s are known. For example, even though the name `operator<<` is known within the definition of `printall()` and a declaration of it can be found in `<iostream>`, the actual declaration of `operator<<` needed to print `p[i]` cannot be known until it is known what type `T` is (14.6.2). — *end example*]

10 If a name does not depend on a *template-parameter* (as defined in 14.6.2), a declaration (or set of declarations) for that name shall be in scope at the point where the name appears in the template definition; the name is bound to the declaration (or declarations) found at that point and this binding is not affected by declarations that are visible at the point of instantiation. [*Example:*

```
void f(char);

template<class T> void g(T t) {
  f(1);             // f(char)
  f(T(1));          // dependent
  f(t);             // dependent
  dd++;             // not dependent
```

```
                        // error: declaration for dd not found
}

enum E { e };
void f(E);

double dd;
void h() {
  g(e);                 // will cause one call of f(char) followed
                        // by two calls of f(E)
  g('a');               // will cause three calls of f(char)
}
```

— *end example*]

11 [*Note:* for purposes of name lookup, default arguments of function templates and default arguments of member functions of class templates are considered definitions (14.5). — *end note*]

14.6.1 Locally declared names [temp.local]

1 Like normal (non-template) classes, class templates have an injected-class-name (Clause 9). The injected-class-name can be used with or without a *template-argument-list*. When it is used without a *template-argument-list*, it is equivalent to the injected-class-name followed by the *template-parameter*s of the class template enclosed in <>. When it is used with a *template-argument-list*, it refers to the specified class template specialization, which could be the current specialization or another specialization.

2 Within the scope of a class template specialization or partial specialization, when the injected-class-name is not followed by a <, it is equivalent to the injected-class-name followed by the *template-argument*s of the class template specialization or partial specialization enclosed in <>. [*Example:*

```
template<class T> class Y;
template<> class Y<int> {
  Y* p;                 // meaning Y<int>
  Y<char>* q;           // meaning Y<char>
};
```

— *end example*]

3 The injected-class-name of a class template or class template specialization can be used either with or without a *template-argument-list* wherever it is in scope. [*Example:*

```
template <class T> struct Base {
  Base* p;
};

template <class T> struct Derived: public Base<T> {
  typename Derived::Base* p;     // meaning Derived::Base<T>
};
```

— *end example*]

4 A lookup that finds an injected-class-name (10.2) can result in an ambiguity in certain cases (for example, if it is found in more than one base class). If all of the injected-class-names that are found refer to specializations of the same class template, and if the name is followed by a *template-argument-list*, the reference refers to the class template itself and not a specialization thereof, and is not ambiguous. [*Example:*

```
template <class T> struct Base { };
template <class T> struct Derived: Base<int>, Base<char> {
  typename Derived::Base b;             // error: ambiguous
  typename Derived::Base<double> d;     // OK
};
```

— *end example*]

5 When the normal name of the template (i.e., the name from the enclosing scope, not the injected-class-name) is used without a *template-argument-list*, it refers to the class template itself and not a specialization of the template. [*Example:*

```
template <class T> class X {
  X* p;             // meaning X<T>
  X<T>* p2;
  X<int>* p3;
  ::X* p4;          // error: missing template argument list
                    // ::X does not refer to the injected-class-name
};
```

— *end example*]

6 The scope of a *template-parameter* extends from its point of declaration until the end of its template. A *template-parameter* hides any entity with the same name in the enclosing scope. [*Note:* this implies that a *template-parameter* can be used in the declaration of subsequent *template-parameter*s and their default arguments but cannot be used in preceding *template-parameter*s or their default arguments. For example,

```
template<class T, T* p, class U = T> class X { /* ... */ };
template<class T> void f(T* p = new T);
```

This also implies that a *template-parameter* can be used in the specification of base classes. For example,

```
template<class T> class X : public Array<T> { /* ... */ };
template<class T> class Y : public T { /* ... */ };
```

The use of a *template-parameter* as a base class implies that a class used as a *template-argument* must be defined and not just declared when the class template is instantiated. — *end note*]

7 A *template-parameter* shall not be redeclared within its scope (including nested scopes). A *template-parameter* shall not have the same name as the template name. [*Example:*

```
template<class T, int i> class Y {
  int T;            // error: template-parameter redeclared
  void f() {
    char T;         // error: template-parameter redeclared
  }
};

template<class X> class X;      // error: template-parameter redeclared
```

— *end example*]

8 In the definition of a member of a class template that appears outside of the class template definition, the name of a member of this template hides the name of a *template-parameter*. [*Example:*

```
template<class T> struct A {
  struct B { /* ... */ };
  void f();
```

```
};

template<class B> void A<B>::f() {
  B b;              // A's B, not the template parameter
}
```

— *end example*]

9 In the definition of a member of a class template that appears outside of the namespace containing the class template definition, the name of a *template-parameter* hides the name of a member of this namespace. [*Example:*

```
namespace N {
  class C { };
  template<class T> class B {
    void f(T);
  };
}
template<class C> void N::B<C>::f(C) {
  C b;              // C is the template parameter, not N::C
}
```

— *end example*]

10 In the definition of a class template or in the definition of a member of such a template that appears outside of the template definition, for each base class which does not depend on a *template-parameter* (14.6.2), if the name of the base class or the name of a member of the base class is the same as the name of a *template-parameter*, the base class name or member name hides the *template-parameter* name (3.3.10). [*Example:*

```
struct A {
  struct B { /* ... */ };
  int a;
  int Y;
};

template<class B, class a> struct X : A {
  B b;              // A's B
  a b;              // error: A's a isn't a type name
};
```

— *end example*]

14.6.2 Dependent names [temp.dep]

1 Inside a template, some constructs have semantics which may differ from one instantiation to another. Such a construct *depends* on the template parameters. In particular, types and expressions may depend on the type and/or value of template parameters (as determined by the template arguments) and this determines the context for name lookup for certain names. Expressions may be *type-dependent* (on the type of a template parameter) or *value-dependent* (on the value of a non-type template parameter). In an expression of the form:

postfix-expression (*expression-list*$_{opt}$)

where the *postfix-expression* is an *unqualified-id* but not a *template-id*, the *unqualified-id* denotes a *dependent name* if and only if any of the expressions in the *expression-list* is a type-dependent expression (14.6.2.2). If an operand of an operator is a type-dependent expression, the operator also denotes a dependent name.

Such names are unbound and are looked up at the point of the template instantiation (14.6.4.1) in both the context of the template definition and the context of the point of instantiation.

2 [*Example:*

```
template<class T> struct X : B<T> {
  typename T::A* pa;
  void f(B<T>* pb) {
    static int i = B<T>::i;
    pb->j++;
  }
};
```

the base class name `B<T>`, the type name `T::A`, the names `B<T>::i` and `pb->j` explicitly depend on the *template-parameter*. — *end example*]

3 In the definition of a class template or a member of a class template, if a base class of the class template depends on a *template-parameter*, the base class scope is not examined during unqualified name lookup either at the point of definition of the class template or member or during an instantiation of the class template or member. [*Example:*

```
typedef double A;
template<class T> class B {
  typedef int A;
};
template<class T> struct X : B<T> {
  A a;              // a has type double
};
```

The type name `A` in the definition of `X<T>` binds to the typedef name defined in the global namespace scope, not to the typedef name defined in the base class `B<T>`. — *end example*] [*Example:*

```
struct A {
  struct B { /* ... */ };
  int a;
  int Y;
};

int a;

template<class T> struct Y : T {
  struct B { /* ... */ };
  B b;                          // The B defined in Y
  void f(int i) { a = i; }      // ::a
  Y* p;                         // Y<T>
};

Y<A> ya;
```

The members `A::B`, `A::a`, and `A::Y` of the template argument `A` do not affect the binding of names in `Y<A>`. — *end example*]

14.6.2.1 Dependent types [temp.dep.type]

1 In the definition of a class template, a nested class of a class template, a member of a class template, or a member of a nested class of a class template, a name refers to the *current instantiation* if it is

— the injected-class-name (9) of the class template or nested class,

— in the definition of a primary class template, the name of the class template followed by the template argument list of the primary template (as described below) enclosed in <>,

— in the definition of a nested class of a class template, the name of the nested class referenced as a member of the current instantiation, or

— in the definition of a partial specialization, the name of the class template followed by the template argument list of the partial specialization enclosed in <>. If the *n*th template parameter is a parameter pack, the *n*th template argument is a pack expansion (14.5.3) whose pattern is the name of the parameter pack.

2 The template argument list of a primary template is a template argument list in which the *n*th template argument has the value of the *n*th template parameter of the class template. If the *n*th template parameter is a template parameter pack, the *n*th template argument is a pack expansion (14.5.3) whose pattern is the name of the template parameter pack.

3 A template argument that is equivalent to a template parameter (i.e., has the same constant value or the same type as the template parameter) can be used in place of that template parameter in a reference to the current instantiation. In the case of a non-type template argument, the argument must have been given the value of the template parameter and not an expression in which the template parameter appears as a subexpression. [*Example:*

```
template <class T> class A {
  A* p1;                          // A is the current instantiation
  A<T>* p2;                       // A<T> is the current instantiation
  A<T*> p3;                       // A<T*> is not the current instantiation
  ::A<T>* p4;                     // ::A<T> is the current instantiation
  class B {
    B* p1;                        // B is the current instantiation
    A<T>::B* p2;                  // A<T>::B is the current instantiation
    typename A<T*>::B* p3;        // A<T*>::B is not the
                                  // current instantiation
  };
};

template <class T> class A<T*> {
  A<T*>* p1;                      // A<T*> is the current instantiation
  A<T>* p2;                       // A<T> is not the current instantiation
};

template <class T1, class T2, int I> struct B {
  B<T1, T2, I>* b1;               // refers to the current instantiation
  B<T2, T1, I>* b2;               // not the current instantiation
  typedef T1 my_T1;
  static const int my_I = I;
  static const int my_I2 = I+0;
  static const int my_I3 = my_I;
  B<my_T1, T2, my_I>* b3;         // refers to the current instantiation
  B<my_T1, T2, my_I2>* b4;        // not the current instantiation
  B<my_T1, T2, my_I3>* b5;        // refers to the current instantiation
};
```

— *end example*]

4 A name is a *member of the current instantiation* if it is

— An unqualified name that, when looked up, refers to a member of a class template. [*Note:* this can only occur when looking up a name in a scope enclosed by the definition of a class template. — *end note*]

— A *qualified-id* in which the *nested-name-specifier* refers to the current instantiation.

[*Example:*

```
template <class T> class A {
  static const int i = 5;
  int n1[i];         // i refers to a member of the current instantiation
  int n2[A::i];      // A::i refers to a member of the current instantiation
  int n3[A<T>::i];   // A<T>::i refers to a member of the current instantiation
  int f();
};

template <class T> int A<T>::f() {
  return i;          // i refers to a member of the current instantiation
}
```

— *end example*]

5 A name is a *member of an unknown specialization* if the name is a *qualified-id* in which the *nested-name-specifier* names a dependent type that is not the current instantiation.

6 A type is dependent if it is

— a template parameter,

— a member of an unknown specialization,

— a nested class that is a member of the current instantiation,

— a cv-qualified type where the cv-unqualified type is dependent,

— a compound type constructed from any dependent type,

— an array type constructed from any dependent type or whose size is specified by a constant expression that is value-dependent,

— a *simple-template-id* in which either the template name is a template parameter or any of the template arguments is a dependent type or an expression that is type-dependent or value-dependent, or

— denoted by `decltype(`*expression* `)`, where *expression* is type-dependent (14.6.2.2).

7 [*Note:* because typedefs do not introduce new types, but instead simply refer to other types, a name that refers to a typedef that is a member of the current instantiation is dependent only if the type referred to is dependent. — *end note*]

14.6.2.2 Type-dependent expressions [temp.dep.expr]

1 Except as described below, an expression is type-dependent if any subexpression is type-dependent.

2 `this` is type-dependent if the class type of the enclosing member function is dependent (14.6.2.1).

3 An *id-expression* is type-dependent if it contains:

— an *identifier* that was declared with a dependent type,

— a *template-id* that is dependent,

— a *conversion-function-id* that specifies a dependent type,

— a *nested-name-specifier* or a *qualified-id* that names a member of an unknown specialization, or

— an *identifier*, *nested-name-specifier*, or a *qualified-id* that names a member of the current instantiation that is a cosntrained member (9.2).

Expressions of the following forms are type-dependent only if the type specified by the *type-id*, *simple-type-specifier* or *new-type-id* is dependent, even if any subexpression is type-dependent:

simple-type-specifier `(` *expression-list*$_{opt}$ `)`
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id* *new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$
`dynamic_cast` `<` *type-id* `>` `(` *expression* `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`(` *type-id* `)` *cast-expression*

4 Expressions of the following forms are never type-dependent (because the type of the expression cannot be dependent):

literal
postfix-expression `.` *pseudo-destructor-name*
postfix-expression `->` *pseudo-destructor-name*
`sizeof` *unary-expression*
`sizeof` `(` *type-id* `)`
`sizeof` `...` `(` *identifier* `)`
`alignof` `(` *type-id* `)`
`typeid` `(` *expression* `)`
`typeid` `(` *type-id* `)`
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete` `[ ]` *cast-expression*
`throw` *assignment-expression*$_{opt}$

[*Note:* For the standard library macro `offsetof`, see 18.1. — *end note*]

5 A class member access expression (5.2.5) is type-dependent if the type of the referenced member is dependent. [*Note:* in an expression of the form `x.y` or `xp->y` the type of the expression is usually the type of the member `y` of the class of `x` (or the class pointed to by `xp`). However, if `x` or `xp` refers to a dependent type that is not the current instantiation, the type of `y` is always dependent. If `x` or `xp` refers to a non-dependent type or refers to the current instantiation, the type of `y` is the type of the class member access expression. — *end note*]

14.6.2.3 Value-dependent expressions [temp.dep.constexpr]

1 Except as described below, a constant expression is value-dependent if any subexpression is value-dependent.

2 An *identifier* is value-dependent if it is:

— a name declared with a dependent type,

— the name of a non-type template parameter,

— a constant with effective literal type and is initialized with an expression that is value-dependent.

Expressions of the following form are value-dependent if the *unary-expression* is type-dependent or the *type-id* is dependent:

`sizeof` *unary-expression*
`sizeof` `(` *type-id* `)`
`alignof` `(` *type-id* `)`

[*Note:* For the standard library macro `offsetof`, see 18.1. — *end note*]

3 Expressions of the following form are value-dependent if either the *type-id* or *simple-type-specifier* is dependent or the *expression* or *cast-expression* is value-dependent:

simple-type-specifier `(` *expression-list*$_{opt}$ `)`
`static_cast` `<` *type-id* `>` `(` *expression* `)`
`const_cast` `<` *type-id* `>` `(` *expression* `)`
`reinterpret_cast` `<` *type-id* `>` `(` *expression* `)`
`(` *type-id* `)` *cast-expression*

4 Expressions of the following form are value-dependent:

`sizeof ...` `(` *identifier* `)`

14.6.2.4 Dependent template arguments [temp.dep.temp]

1 A type *template-argument* is dependent if the type it specifies is dependent.

2 An integral non-type *template-argument* is dependent if the constant expression it specifies is value-dependent.

3 A non-integral non-type *template-argument* is dependent if its type is dependent or it has either of the following forms

qualified-id
`&` *qualified-id*

and contains a *nested-name-specifier* which specifies a *class-name* that names a dependent type.

4 A template *template-argument* is dependent if it names a *template-parameter* or is a *qualified-id* with a *nested-name-specifier* which contains a *class-name* that names a dependent type.

14.6.3 Non-dependent names [temp.nondep]

1 Non-dependent names used in a template definition are found using the usual name lookup and bound at the point they are used. [*Example:*

```
void g(double);
void h();

template<class T> class Z {
public:
  void f() {
    g(1);           // calls g(double)
    h++;            // ill-formed: cannot increment function;
                    // this could be diagnosed either here or
                    // at the point of instantiation
  }
};

void g(int);        // not in scope at the point of the template
                    // definition, not considered for the call g(1)
```

— *end example*]

2 [*Note:* if a template contains template requirements, name lookup of non-dependent names in its constrained contexts (14.10) can find the names of associated functions in the requirements scope (3.3.8). — *end note*]

14.6.4 Dependent name resolution [temp.dep.res]

1 In resolving dependent names, names from the following sources are considered:

— Declarations that are visible at the point of definition of the template.

— Declarations from namespaces associated with the types of the function arguments both from the instantiation context (14.6.4.1) and from the definition context.

14.6.4.1 Point of instantiation [temp.point]

1 For a function template specialization, a member function template specialization, or a specialization for a member function or static data member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization and the context from which it is referenced depends on a template parameter, the point of instantiation of the specialization is the point of instantiation of the enclosing specialization. Otherwise, the point of instantiation for such a specialization immediately follows the namespace scope declaration or definition that refers to the specialization.

2 If a function template or member function of a class template is called in a way which uses the definition of a default argument of that function template or member function, the point of instantiation of the default argument is the point of instantiation of the function template or member function specialization.

3 For a class template specialization, a class member template specialization, or a specialization for a class member of a class template, if the specialization is implicitly instantiated because it is referenced from within another template specialization, if the context from which the specialization is referenced depends on a template parameter, and if the specialization is not instantiated previous to the instantiation of the enclosing template, the point of instantiation is immediately before the point of instantiation of the enclosing template. Otherwise, the point of instantiation for such a specialization immediately precedes the namespace scope declaration or definition that refers to the specialization.

4 If a virtual function is implicitly instantiated, its point of instantiation is immediately following the point of instantiation of its enclosing class template specialization.

5 An explicit instantiation definition is an instantiation point for the specialization or specializations specified by the explicit instantiation.

6 The instantiation context of an expression that depends on the template arguments is the set of declarations with external linkage declared prior to the point of instantiation of the template specialization in the same translation unit.

7 A specialization for a function template, a member function template, or of a member function or static data member of a class template may have multiple points of instantiations within a translation unit. A specialization for a class template has at most one point of instantiation within a translation unit. A specialization for any template may have points of instantiation in multiple translation units. If two different points of instantiation give a template specialization different meanings according to the one definition rule (3.2), the program is ill-formed, no diagnostic required.

14.6.4.2 Candidate functions [temp.dep.candidate]

1 For a function call that depends on a template parameter, if the function name is an *unqualified-id* but not a *template-id*, or if the function is called using operator notation, the candidate functions are found using the usual lookup rules (3.4.1, 3.4.2) except that:

— For the part of the lookup using unqualified name lookup (3.4.1), only function declarations with external linkage from the template definition context are found.

— For the part of the lookup using associated namespaces (3.4.2), only function declarations with external linkage found in either the template definition context or the template instantiation context are found.

If the call would be ill-formed or would find a better match had the lookup within the associated namespaces considered all the function declarations with external linkage introduced in those namespaces in all translation units, not just considering those declarations found in the template definition and template instantiation contexts, then the program has undefined behavior.

14.6.5 Friend names declared within a class template [temp.inject]

1 Friend classes or functions can be declared within a class template. When a template is instantiated, the names of its friends are treated as if the specialization had been explicitly declared at its point of instantiation.

2 As with non-template classes, the names of namespace-scope friend functions of a class template specialization are not visible during an ordinary lookup unless explicitly declared at namespace scope (11.4). Such names may be found under the rules for associated classes (3.4.2).[135] [*Example:*

```
template<typename T> struct number {
  number(int);
  friend number gcd(number x, number y) { return 0; };
};

void g() {
  number<double> a(3), b(4);
  a = gcd(a,b);      // finds gcd because number<double> is an
                     // associated class, making gcd visible
                     // in its namespace (global scope)
  b = gcd(3,4);      // ill-formed; gcd is not visible
}
```

— *end example*]

14.7 Template instantiation and specialization [temp.spec]

1 The act of instantiating a function, a class, a concept map, a member of a class template or a member template is referred to as *template instantiation*.

2 A function instantiated from a function template is called an instantiated function. A class instantiated from a class template is called an instantiated class. A concept map instantiated from a concept map template is called an instantiated concept map. A member function, a member class, or a static data member of a class template instantiated from the member definition of the class template is called, respectively, an instantiated member function, member class or static data member. A member function instantiated from a member function template is called an instantiated member function. A member class instantiated from a member class template is called an instantiated member class.

3 An explicit specialization may be declared for a function template, a class template, a member of a class template or a member template. An explicit specialization declaration is introduced by `template<>`. In an explicit specialization declaration for a class template, a member of a class template or a class member template, the name of the class that is explicitly specialized shall be a *simple-template-id*. In the explicit

135) Friend declarations do not introduce new names into any scope, either when the template is declared or when it is instantiated.

specialization declaration for a function template or a member function template, the name of the function or member function explicitly specialized may be a *template-id*. [*Example:*

```
template<class T = int> struct A {
  static int x;
};
template<class U> void g(U) { }

template<> struct A<double> { };        // specialize for T == double
template<> struct A<> { };              // specialize for T == int
template<> void g(char) { }             // specialize for U == char
                                        // U is deduced from the parameter type
template<> void g<int>(int) { }         // specialize for U == int
template<> int A<char>::x = 0;          // specialize for T == char

template<class T = int> struct B {
  static int x;
};
template<> int B<>::x = 1;              // specialize for T == int
```

— *end example*]

4 An instantiated template specialization can be either implicitly instantiated (14.7.1) for a given argument list or be explicitly instantiated (14.7.2). A specialization is a class, function, or class member that is either instantiated or explicitly specialized (14.7.3).

5 For a given template and a given set of *template-argument*s,

— an explicit instantiation definition shall appear at most once in a program,

— an explicit specialization shall be defined at most once in a program (according to 3.2), and

— both an explicit instantiation and a declaration of an explicit specialization shall not appear in a program unless the explicit instantiation follows a declaration of the explicit specialization.

An implementation is not required to diagnose a violation of this rule.

6 Each class template specialization instantiated from a template has its own copy of any static members. [*Example:*

```
template<class T> class X {
  static T s;
};
template<class T> T X<T>::s = 0;
X<int> aa;
X<char*> bb;
```

`X<int>` has a static member `s` of type `int` and `X<char*>` has a static member `s` of type `char*`. — *end example*]

14.7.1 Implicit instantiation [temp.inst]

1 Unless a class template specialization has been explicitly instantiated (14.7.2) or explicitly specialized (14.7.3), the class template specialization is implicitly instantiated when the specialization is referenced in a context that requires a completely-defined object type or when the completeness of the class type affects the semantics of the program. The implicit instantiation of a class template specialization causes the implicit

instantiation of the declarations, but not of the definitions or default arguments, of the class member functions, member classes, static data members and member templates; and it causes the implicit instantiation of the definitions of member anonymous unions. Unless a member of a class template or a member template has been explicitly instantiated or explicitly specialized, the specialization of the member is implicitly instantiated when the specialization is referenced in a context that requires the member definition to exist; in particular, the initialization (and any associated side-effects) of a static data member does not occur unless the static data member is itself used in a way that requires the definition of the static data member to exist.

2 Unless a function template specialization has been explicitly instantiated or explicitly specialized, the function template specialization is implicitly instantiated when the specialization is referenced in a context that requires a function definition to exist. Unless a call is to a function template explicit specialization or to a member function of an explicitly specialized class template, a default argument for a function template or a member function of a class template is implicitly instantiated when the function is called in a context that requires the value of the default argument.

3 [*Example:*

```
template<class T> struct Z {
  void f();
  void g();
};

void h() {
  Z<int> a;          // instantiation of class Z<int> required
  Z<char>* p;        // instantiation of class Z<char> not required
  Z<double>* q;      // instantiation of class Z<double> not required

  a.f();             // instantiation of Z<int>::f() required
  p->g();            // instantiation of class Z<char> required, and
                     // instantiation of Z<char>::g() required
}
```

Nothing in this example requires `class Z<double>`, `Z<int>::g()`, or `Z<char>::f()` to be implicitly instantiated. — *end example*]

4 A class template specialization is implicitly instantiated if the class type is used in a context that requires a completely-defined object type or if the completeness of the class type might affect the semantics of the program. [*Note:* in particular, if the semantics of an expression depend on the member or base class lists of a class template specialization, the class template specialization is implicitly generated. For instance, deleting a pointer to class type depends on whether or not the class declares a destructor, and conversion between pointer to class types depends on the inheritance relationship between the two classes involved. — *end note*] [*Example:*

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

void f(void*);
void f(B<int>*);

void g(D<int>* p, D<char>* pp, D<double>* ppp) {
  f(p);              // instantiation of D<int> required: call f(B<int>*)
  B<char>* q = pp;   // instantiation of D<char> required:
                     // convert D<char>* to B<char>*
  delete ppp;        // instantiation of D<double> required
}
```

— *end example*]

5 If the overload resolution process can determine the correct function to call without instantiating a class template definition or concept map template definition, it is unspecified whether that instantiation actually takes place. [*Example:*

```
template <class T> struct S {
  operator int();
};

void f(int);
void f(S<int>&);
void f(S<float>);

void g(S<int>& sr) {
  f(sr);            // instantiation of S<int> allowed but not required
                    // instantiation of S<float> allowed but not required
};
```

— *end example*]

6 If an implicit instantiation of a class template specialization is required and the template is declared but not defined, the program is ill-formed. [*Example:*

```
template<class T> class X;

X<char> ch;         // error: definition of X required
```

— *end example*]

7 The implicit instantiation of a class template does not cause any static data members of that class to be implicitly instantiated.

8 If a function template or a member function template specialization is used in a way that involves overload resolution, a declaration of the specialization is implicitly instantiated (14.8.3).

9 An implementation shall not implicitly instantiate a function template, a member template, a non-virtual member function, a concept map template, a member class, or a static data member of a class template that does not require instantiation. It is unspecified whether or not an implementation implicitly instantiates a virtual member function of a class template if the virtual member function would not otherwise be instantiated. The use of a template specialization in a default argument shall not cause the template to be implicitly instantiated except that a class template may be instantiated where its complete type is needed to determine the correctness of the default argument. The use of a default argument in a function call causes specializations in the default argument to be implicitly instantiated.

10 Implicitly instantiated class, concept map, and function template specializations are placed in the namespace where the template is defined. Implicitly instantiated specializations for members of a class template are placed in the namespace where the enclosing class template is defined. Implicitly instantiated member templates are placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
  template<class T> class List {
  public:
    T* get();
  };
}
```

```
template<class K, class V> class Map {
  N::List<V> lt;
  V get(K);
};

void g(Map<char*,int>& m) {
  int i = m.get("Nicholas");
}
```

a call of `lt.get()` from `Map<char*,int>::get()` would place `List<int>::get()` in the namespace `N` rather than in the global namespace. — *end example*]

11 If a function template `f` is called in a way that requires a default argument expression to be used, the dependent names are looked up, the semantics constraints are checked, and the instantiation of any template used in the default argument expression is done as if the default argument expression had been an expression used in a function template specialization with the same scope, the same template parameters and the same access as that of the function template `f` used at that point. This analysis is called *default argument instantiation*. The instantiated default argument is then used as the argument of `f`.

12 Each default argument is instantiated independently. [*Example:*

```
template<class T> void f(T x, T y = ydef(T()), T z = zdef(T()));

class  A { };

A zdef(A);

void g(A a, A b, A c) {
  f(a, b, c);       // no default argument instantiation
  f(a, b);          // default argument z = zdef(T()) instantiated
  f(a);             // ill-formed; ydef is not declared
}
```

— *end example*]

13 [*Note:* 14.6.4.1 defines the point of instantiation of a template specialization. — *end note*]

14 There is an implementation-defined quantity that specifies the limit on the total depth of recursive instantiations, which could involve more than one template. The result of an infinite recursion in instantiation is undefined. [*Example:*

```
template<class T> class X {
  X<T>* p;          // OK
  X<T*> a;          // implicit generation of X<T> requires
                    // the implicit instantiation of X<T*> which requires
                    // the implicit instantiation of X<T**> which ...
};
```

— *end example*]

15 If no concept map exists for a given concept instance, and there exists a concept map template that matches the concept instance, the concept map is implicitly instantiated when the concept map is referenced in a

context that requires the concept map definition, either to satisfy a concept requirement (14.10.1) or when the *nested-name-specifier* of a *qualified-id* references a concept instance (3.4.3.3).

14.7.2 Explicit instantiation [temp.explicit]

1 A class, a function or member template specialization can be explicitly instantiated from its template. A member function, member class or static data member of a class template can be explicitly instantiated from the member definition associated with its class template. An explicit instantiation of a function template shall not use the `inline` or `constexpr` specifiers.

2 The syntax for explicit instantiation is:

explicit-instantiation:
`extern`$_{opt}$ `template` *declaration*

There are two forms of explicit instantiation: an explicit instantiation definition and an explicit instantiation declaration. An explicit instantiation declaration begins with the `extern` keyword.

If the explicit instantiation is for a class or member class, the *elaborated-type-specifier* in the *declaration* shall include a *simple-template-id*. If the explicit instantiation is for a function or member function, the *unqualified-id* in the *declaration* shall be either a *template-id* or, where all template arguments can be deduced, a *template-name* or *operator-function-id*. [*Note:* the declaration may declare a *qualified-id*, in which case the *unqualified-id* of the *qualified-id* must be a *template-id*. — *end note*] If the explicit instantiation is for a member function, a member class or a static data member of a class template specialization, the name of the class template specialization in the *qualified-id* for the member name shall be a *simple-template-id*. An explicit instantiation shall appear in an enclosing namespace of its template. If the name declared in the explicit instantiation is an unqualified name, the explicit instantiation shall appear in the namespace where its template is declared or, if that namespace is inline (7.3.1), any namespace from its enclosing namespace set. [*Note:* regarding qualified names in declarators, see 8.3. — *end note*] [*Example:*

```
template<class T> class Array { void mf(); };
template class Array<char>;
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);       // argument is deduced here

namespace N {
  template<class T> void f(T&) { }
}
template void N::f<int>(int&);
```

— *end example*]

3 A declaration of a function template shall be in scope at the point of the explicit instantiation of the function template. A definition of the class or class template containing a member function template shall be in scope at the point of the explicit instantiation of the member function template. A definition of a class template or class member template shall be in scope at the point of the explicit instantiation of the class template or class member template. A definition of a class template shall be in scope at the point of an explicit instantiation of a member function or a static data member of the class template. A definition of a member class of a class template shall be in scope at the point of an explicit instantiation of the member class. If the *declaration* of the explicit instantiation names an implicitly-declared special member function (Clause 12), the program is ill-formed.

4 For a given set of template parameters, if an explicit instantiation of a template appears after a declaration of an explicit specialization for that template, the explicit instantiation has no effect. Otherwise, for an

explicit instantiation definition the definition of a non-exported function template, a non-exported member function template, or a non-exported member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

5 An explicit instantiation of a class or function template specialization is placed in the namespace in which the template is defined. An explicit instantiation for a member of a class template is placed in the namespace where the enclosing class template is defined. An explicit instantiation for a member template is placed in the namespace where the enclosing class or class template is defined. [*Example:*

```
namespace N {
  template<class T> class Y { void mf() { } };
}

template class Y<int>;                  // error: class template Y not visible
                                        // in the global namespace

using N::Y;
template class Y<int>;                  // OK: explicit instantiation in namespace N

template class N::Y<char*>;             // OK: explicit instantiation in namespace N
template void N::Y<double>::mf();       // OK: explicit instantiation
                                        // in namespace N
```

— *end example*]

6 A trailing *template-argument* can be left unspecified in an explicit instantiation of a function template specialization or of a member function template specialization provided it can be deduced from the type of a function parameter (14.8.2). [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v);

// instantiate sort(Array<int>&) - template-argument deduced
template void sort<>(Array<int>&);
```

— *end example*]

7 An explicit instantiation that names a class template specialization is an explicit instantion of the same kind (declaration or definition) of each of its members (not including members inherited from base classes) that has not been previously explicitly specialized in the translation unit containing the explicit instantiation, except as described below.

8 An explicit instantiation definition that names a class template specialization explicitly instantiates the class template specialization and is only an explicit instantiation definition of members whose definition is visible at the point of instantiation.

9 An explicit instantiation declaration that names a class template specialization has no effect on the class template specialization itself (except for perhaps resulting in its implicit instantiation). Except for inline functions, other explicit instantiation declarations have the effect of suppressing the implicit instantiation of the entity to which they refer. [*Note:* The intent is that an inline function that is the subject of an explicit instantiation declaration will still be implicitly instantiated when used so that the body can be considered for inlining, but that no out-of-line copy of the inline function would be generated in the translation unit. — *end note*]

10 If an entity is the subject of both an explicit instantiation declaration and an explicit instantiation definition in the same translation unit, the definition shall follow the declaration. An entity that is the subject of

an explicit instantiation declaration and that is also used in the translation unit shall be the subject of an explicit instantiation definition somewhere in the program; otherwise the program is ill-formed, no diagnostic required. [*Note:* This rule does apply to inline functions even though an explicit instantiation declaration of such an entity has no other normative effect. This is needed to ensure that if the address of an inline function is taken in a translation unit in which the implementation chose to suppress the out-of-line body, another translation unit will supply the body. — *end note*] An explicit instantiation declaration shall not name a specialization of a template with internal linkage.

11 The usual access checking rules do not apply to names used to specify explicit instantiations. [*Note:* In particular, the template arguments and names used in the function declarator (including parameter types, return types and exception specifications) may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible. — *end note*]

12 An explicit instantiation does not constitute a use of a default argument, so default argument instantiation is not done. [*Example:*

```
char* p = 0;
template<class T> T g(T = &p);
template int g<int>(int);       // OK even though &p isn't an int.
```

— *end example*]

14.7.3 Explicit specialization [temp.expl.spec]

1 An explicit specialization of any of the following:

- non-deleted function template
- class template
- non-deleted member function of a class template
- static data member of a class template
- member class of a class template
- member class template of a class template
- non-deleted member function template of a class template

can be declared by a declaration introduced by `template<>`; that is:

explicit-specialization:
`template < >` *declaration*

[*Example:*

```
template<class T> class stream;

template<> class stream<char> { /* ... */ };

template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

template<> void sort<char*>(Array<char*>&) ;
```

Given these declarations, `stream<char>` will be used as the definition of streams of `char`s; other streams will be handled by class template specializations instantiated from the class template. Similarly, `sort<char*>`

will be used as the sort function for arguments of type `Array<char*>`; other `Array` types will be sorted by functions generated from the template. — *end example*]

2 An explicit specialization shall be declared in the nearest enclosing namespace of the template, or, if the namespace is inline (7.3.1), any namespace from its enclosing namespace set. Such a declaration may also be a definition. If the declaration is not a definition, the specialization may be defined later (7.3.1.2).

3 A declaration of a function template or class template being explicitly specialized shall be in scope at the point of declaration of an explicit specialization. [*Note:* a declaration, but not a definition of the template is required. — *end note*] The definition of a class or class template shall be in scope at the point of declaration of an explicit specialization for a member template of the class or class template. [*Example:*

```
template<> class X<int> { /* ... */ };          // error: X not a template

template<class T> class X;

template<> class X<char*> { /* ... */ };        // OK: X is a template
```

— *end example*]

4 A member function, a member class or a static data member of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; in this case, the definition of the class template shall be in scope at the point of declaration of the explicit specialization for the member of the class template. If such an explicit specialization for the member of a class template names an implicitly-declared special member function (Clause 12), the program is ill-formed.

5 A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined. In this case, the definition of the class template explicit specialization shall be in scope at the point of declaration of the explicit specialization of the member. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization. Definitions of members of an explicitly specialized class are defined in the same manner as members of normal classes, and not using the syntax for explicit specialization. [*Example:*

```
template<class T> struct A {
  void f(T) { /* ... */ }
};

template<> struct A<int> {
  void f(int);
};

void h() {
  A<int> a;
  a.f(16);          // A<int>::f must be defined somewhere
}

// explicit specialization syntax not used for a member of
// explicitly specialized class template specialization
void A<int>::f(int) { /* ... */ }
```

— *end example*]

6 If a template, a member template or the member of a class template is explicitly specialized then that specialization shall be declared before the first use of that specialization that would cause an implicit instan-

tiation to take place, in every translation unit in which such a use occurs; no diagnostic is required. If the program does not provide a definition for an explicit specialization and either the specialization is used in a way that would cause an implicit instantiation to take place or the member is a virtual member function, the program is ill-formed, no diagnostic required. An implicit instantiation is never generated for an explicit specialization that is declared but not defined. [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v) { /* ... */ }

void f(Array<String>& v) {
  sort(v);          // use primary template
                    // sort(Array<T>&), T is String
}

template<> void sort<String>(Array<String>& v); // error: specialization
                                                // after use of primary template
template<> void sort<>(Array<char*>& v);        // OK: sort<char*> not yet used
```

— *end example*]

7 The placement of explicit specialization declarations for function templates, class templates, member functions of class templates, static data members of class templates, member classes of class templates, member class templates of class templates, member function templates of class templates, member functions of member templates of class templates, member functions of member templates of non-template classes, member function templates of member classes of class templates, etc., and the placement of partial specialization declarations of class templates, member class templates of non-template classes, member class templates of class templates, etc., can affect whether a program is well-formed according to the relative positioning of the explicit specialization declarations and their points of instantiation in the translation unit as specified above and below. When writing a specialization, be careful about its location; or to make it compile will be such a trial as to kindle its self-immolation.

8 When a specialization for which an explicit specialization exists is used within the instantiation of an exported template, and the unspecialized template name is non-dependent in the exported template, a declaration of the explicit specialization shall be declared before the definition of the exported template, in the translation unit containing that definition. [*Example:*

```
// file #1
#include <vector>
// Primary class template vector
export template<class T> void f(t) {
  std::vector<T> vec;           // should match the specialization
  /* ... */
}

// file #2
#include <vector>
class B { };
// Explicit specialization of vector for vector<B>
namespace std {
  template<> class vector<B> { /* ... */ };
}
template<class T> void f(T);
void g(B b) {
  f(b);             // ill-formed:
                    // f<B> should refer to vector<B>, but the
```

```
                        // specialization was not declared with the
                        // definition of f in file #1
}
```

— *end example*]

9 A template explicit specialization is in the scope of the namespace in which the template was defined. [*Example:*

```
namespace N {
  template<class T> class X { /* ... */ };
  template<class T> class Y { /* ... */ };

  template<> class X<int> { /* ... */ };          // OK: specialization
                                                    // in same namespace
  template<> class Y<double>;                     // forward declare intent to
                                                    // specialize for double
}

template<> class N::Y<double> { /* ... */ };      // OK: specialization
                                                    // in same namespace
```

— *end example*]

10 A *simple-template-id* that names a class template explicit specialization that has been declared but not defined can be used exactly like the names of other incompletely-defined classes (3.9). [*Example:*

```
template<class T> class X;        // X is a class template
template<> class X<int>;

X<int>* p;                        // OK: pointer to declared class X<int>
X<int> x;                         // error: object of incomplete class X<int>
```

— *end example*]

11 A trailing *template-argument* can be left unspecified in the *template-id* naming an explicit function template specialization provided it can be deduced from the function argument type. [*Example:*

```
template<class T> class Array { /* ... */ };
template<class T> void sort(Array<T>& v);

// explicit specialization for sort(Array<int>&)
// with deduced template-argument of type int
template<> void sort(Array<int>&);
```

— *end example*]

12 [*Note:* This paragraph is intentionally empty. — *end note*]

13 A function with the same name as a template and a type that exactly matches that of a template specialization is not an explicit specialization (14.5.6).

14 An explicit specialization of a function template is inline only if it is explicitly declared to be, and independently of whether its function template is. [*Example:*

```
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }
```

```
template<> inline void f<>(int) { /* ... */ }    // OK: inline
template<> int g<>(int) { /* ... */ }            // OK: not inline
```

— *end example*]

15 An explicit specialization of a static data member of a template is a definition if the declaration includes an initializer; otherwise, it is a declaration. [*Note:* there is no syntax for the definition of a static data member of a template that requires default initialization.

```
template<> X Q<int>::x;
```

16 This is a declaration regardless of whether X can be default initialized (8.5). — *end note*]

17 A member or a member template of a class template may be explicitly specialized for a given implicit instantiation of the class template, even if the member or member template is defined in the class template definition. An explicit specialization of a member or member template is specified using the syntax for explicit specialization. [*Example:*

```
template<class T> struct A {
  void f(T);
  template<class X1> void g1(T, X1);
  template<class X2> void g2(T, X2);
  void h(T) { }
};

// specialization
template<> void A<int>::f(int);

// out of class member template definition
template<class T> template<class X1> void A<T>::g1(T, X1) { }

// member template specialization
template<> template<class X1> void A<int>::g1(int, X1);

//member template specialization
template<> template<>
  void A<int>::g1(int, char);           // X1 deduced as char
template<> template<>
  void A<int>::g2<char>(int, char);     // X2 specified as char

// member specialization even if defined in class definition
template<> void A<int>::h(int) { }
```

— *end example*]

18 A member or a member template may be nested within many enclosing class templates. In an explicit specialization for such a member, the member declaration shall be preceded by a `template<>` for each enclosing class template that is explicitly specialized. [*Example:*

```
template<class T1> class A {
  template<class T2> class B {
    void mf();
  };
};
template<> template<> class A<int>::B<double>;
template<> template<> void A<char>::B<char>::mf();
```

— *end example*]

19 In an explicit specialization declaration for a member of a class template or a member template that appears in namespace scope, the member template and some of its enclosing class templates may remain unspecialized, except that the declaration shall not explicitly specialize a class member template if its enclosing class templates are not explicitly specialized as well. In such explicit specialization declaration, the keyword `template` followed by a *template-parameter-list* shall be provided instead of the `template<>` preceding the explicit specialization declaration of the member. The types of the *template-parameters* in the *template-parameter-list* shall be the same as those specified in the primary template definition. [*Example:*

```
template <class T1> class A {
  template<class T2> class B {
    template<class T3> void mf1(T3);
    void mf2();
  };
};
template <> template <class X>
  class A<int>::B {
      template <class T> void mf1(T);
  };
template <> template <> template<class T>
  void A<int>::B<double>::mf1(T t) { }
template <class Y> template <>
  void A<Y>::B<double>::mf2() { }       // ill-formed; B<double> is specialized but
                                        // its enclosing class template A is not
```

— *end example*]

20 A specialization of a member function template or member class template of a non-specialized class template is itself a template.

21 An explicit specialization declaration shall not be a friend declaration.

22 Default function arguments shall not be specified in a declaration or a definition for one of the following explicit specializations:

— the explicit specialization of a function template;

— the explicit specialization of a member function template;

— the explicit specialization of a member function of a class template where the class template specialization to which the member function specialization belongs is implicitly instantiated. [*Note:* default function arguments may be specified in the declaration or definition of a member function of a class template specialization that is explicitly specialized. — *end note*]

23 [*Note:* The template arguments provided for an explicit specialization shall satisfy the template requirements of the primary template (14.5.5.1). [*Example:*

```
concept C<typename T> { }
concept_map C<float> { }

template<typename T> requires C<T> void f(T);

template<> void f<float>(float);// OK: concept_map C<float> satisfies requirement
template<> void f<int>(int);    // ill-formed: no concept map satisfies the requirement for C<int>
```

— *end example*] — *end note*]

14.8 Function template specializations [temp.fct.spec]

1 A function instantiated from a function template is called a function template specialization; so is an explicit specialization of a function template. Template arguments can be explicitly specified when naming the function template specialization, deduced from the context (e.g., deduced from the function arguments in a call to the function template specialization, see 14.8.2), or obtained from default template arguments.

2 Each function template specialization instantiated from a template has its own copy of any static variable. [*Example:*

```
template<class T> void f(T* p) {
  static T s;
};

void g(int a, char* b) {
  f(&a);            // calls f<int>(int*)
  f(&b);            // calls f<char*>(char**)
}
```

Here `f<int>(int*)` has a static variable `s` of type `int` and `f<char*>(char**)` has a static variable `s` of type `char*`. — *end example*]

14.8.1 Explicit template argument specification [temp.arg.explicit]

1 Template arguments can be specified when referring to a function template specialization by qualifying the function template name with the list of *template-argument*s in the same way as *template-argument*s are specified in uses of a class template specialization. [*Example:*

```
template<class T> void sort(Array<T>& v);
void f(Array<dcomplex>& cv, Array<int>& ci) {
  sort<dcomplex>(cv);           // sort(Array<dcomplex>&)
  sort<int>(ci);                // sort(Array<int>&)
}
```

and

```
template<class U, class V> U convert(V v);

void g(double d) {
  int i = convert<int,double>(d);       // int convert(double)
  char c = convert<char,double>(d);     // char convert(double)
}
```

— *end example*]

2 A template argument list may be specified when referring to a specialization of a function template

— when a function is called,

— when the address of a function is taken, when a function initializes a reference to function, or when a pointer to member function is formed,

— in an explicit specialization,

— in an explicit instantiation, or

— in a friend declaration.

3 Trailing template arguments that can be deduced (14.8.2) or obtained from default *template-argument*s may be omitted from the list of explicit *template-argument*s. A trailing template parameter pack not otherwise deduced will be deduced to an empty sequence of template arguments. If all of the template arguments can be deduced, they may all be omitted; in this case, the empty template argument list `<>` itself may also be omitted. In contexts where deduction is done and fails, or in contexts where deduction is not done, if a template argument list is specified and it, along with any default template arguments, identifies a single function template specialization, then the *template-id* is an lvalue for the function template specialization. [*Example:*

```
template<class X, class Y> X f(Y);
template<class X, class Y, class ... Z> X g(Y);
void h() {
  int i = f<int>(5.6);          // Y is deduced to be double; Z is deduced to be an empty sequence
  int j = f(5.6);               // ill-formed: X cannot be deduced
  f<void>(f<int, bool>);        // Y for outer f deduced to be
                                // int (*)(bool); Z is deduced to be an empty sequence
  f<void>(f<int>);              // ill-formed: f<int> does not denote a
                                // single function template specialization
  int k = g<int>(5.6);          // Y is deduced to be double, Z is deduced to an empty sequence
  f<void>(g<int, bool>);        // Y for outer f is deduced to be
                                // int (*)(bool), Z is deduced to an empty sequence
}
```

— *end example*]

4 [*Note:* An empty template argument list can be used to indicate that a given use refers to a specialization of a function template even when a normal (i.e., non-template) function is visible that would otherwise be used. For example:

```
template <class T> int f(T);    // #1
int f(int);                     // #2
int k = f(1);                   // uses #2
int l = f<>(1);                 // uses #1
```

— *end note*]

5 Template arguments that are present shall be specified in the declaration order of their corresponding *template-parameter*s. The template argument list shall not specify more *template-argument*s than there are corresponding *template-parameter*s unless one of the *template-parameter*s is a template parameter pack. [*Example:*

```
template<class X, class Y, class Z> X f(Y,Z);
template<class ... Args> void f2();
void g() {
  f<int,char*,double>("aa",3.0);
  f<int,char*>("aa",3.0);       // Z is deduced to be double
  f<int>("aa",3.0);             // Y is deduced to be const char*, and
                                // Z is deduced to be double
  f("aa",3.0);                  // error: X cannot be deduced
  f2<char, short, int, long>(); // OK
}
```

— *end example*]

6 Implicit conversions (Clause 4) will be performed on a function argument to convert it to the type of the corresponding function parameter if the parameter type contains no *template-parameter*s that participate

in template argument deduction. [*Note:* template parameters do not participate in template argument deduction if they are explicitly specified. For example,

```
template<class T> void f(T);

class Complex {
  Complex(double);
};

void g() {
  f<Complex>(1);                // OK, means f<Complex>(Complex(1))
}
```

— *end note*]

7 [*Note:* because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using a function name, there is no way to provide an explicit template argument list for these function templates. — *end note*]

8 [*Note:* For simple function names, argument dependent lookup (3.4.2) applies even when the function name is not visible within the scope of the call. This is because the call still has the syntactic form of a function call (3.4.1). But when a function template with explicit template arguments is used, the call does not have the correct syntactic form unless there is a function template with that name visible at the point of the call. If no such name is visible, the call is not syntactically well-formed and argument-dependent lookup does not apply. If some such name is visible, argument dependent lookup applies and additional function templates may be found in other namespaces. [*Example:*

```
namespace A {
  struct B { };
  template<int X> void f(B);
}
namespace C {
  template<class T> void f(T t);
}
void g(A::B b) {
  f<3>(b);                      // ill-formed: not a function call
  A::f<3>(b);                   // well-formed
  C::f<3>(b);                   // ill-formed; argument dependent lookup
                                // applies only to unqualified names
  using C::f;
  f<3>(b);                      // well-formed because C::f is visible; then
                                // A::f is found by argument dependent lookup
}
```

— *end example*] — *end note*]

9 Template argument deduction can extend the sequence of template arguments corresponding to a template parameter pack, even when the sequence contains explicitly specified template arguments. [*Example:*

```
template<class ... Types> void f(Types ... values);

void g() {
  f<int*, float*>(0, 0, 0);     // Types is deduced to the sequence int*, float*, int
}
```

— *end example*]

14.8.2 Template argument deduction [temp.deduct]

1 When a function template specialization is referenced, all of the template arguments shall have values. The values can be explicitly specified or, in some cases, be deduced from the use or obtained from default *template-argument*s. [*Example:*

```
void f(Array<dcomplex>& cv, Array<int>& ci) {
  sort(cv);                     // calls sort(Array<dcomplex>&)
  sort(ci);                     // calls sort(Array<int>&)
}
```

and

```
void g(double d) {
  int i = convert<int>(d);      // calls convert<int,double>(double)
  int c = convert<char>(d);     // calls convert<char,double>(double)
}
```

— *end example*]

2 When an explicit template argument list is specified, the template arguments must be compatible with the template parameter list and must result in a valid function type as described below; otherwise type deduction fails. Specifically, the following steps are performed when evaluating an explicitly specified template argument list with respect to a given function template:

— The specified template arguments must match the template parameters in kind (i.e., type, non-type, template). There must not be more arguments than there are parameters unless at least one parameter is a template parameter pack, and there shall be an argument for each non-pack parameter. Otherwise, type deduction fails.

— Non-type arguments must match the types of the corresponding non-type template parameters, or must be convertible to the types of the corresponding non-type parameters as specified in 14.3.2, otherwise type deduction fails.

— The specified template argument values are substituted for the corresponding template parameters as specified below.

3 After this substitution is performed, the function parameter type adjustments described in 8.3.5 are performed. [*Example:* A parameter type of "`void ()(const int, int[5])`" becomes "`void(*)(int,int*)`". — *end example*] [*Note:* A top-level qualifier in a function parameter declaration does not affect the function type but still affects the type of the function parameter variable within the function. — *end note*] [*Example:*

```
template <class T> void f(T t);
template <class X> void g(const X x);
template <class Z> void h(Z, Z*);

int main() {
  // #1: function type is f(int), t is non const
  f<int>(1);

  // #2: function type is f(int), t is const
  f<const int>(1);

  // #3: function type is g(int), x is const
  g<int>(1);

  // #4: function type is g(int), x is const
```

```
  g<const int>(1);

  // #5: function type is h(int, const int*)
  h<const int>(1,0);
}
```

— *end example*]

4 [*Note:* `f<int>(1)` and `f<const int>(1)` call distinct functions even though both of the functions called have the same function type. — *end note*]

5 The resulting substituted and adjusted function type is used as the type of the function template for template argument deduction. If a template argument has not been deduced, its default template argument, if any, is used. [*Example:*

```
template <class T, class U = double>
void f(T t = 0, U u = 0);

void g() {
  f(1, 'c');         // f<int,char>(1,'c')
  f(1);              // f<int,double>(1,0)
  f();               // error: T cannot be deduced
  f<int>();          // f<int,double>(0,0)
  f<int,char>();     // f<int,char>(0,0)
}
```

— *end example*]

When all template arguments have been deduced or obtained from default template arguments, all uses of template parameters in non-deduced contexts are replaced with the corresponding deduced or default argument values. If the substitution results in an invalid type, as described above, type deduction fails.

6 At certain points in the template argument deduction process it is is necessary to take a function type that makes use of template parameters and replace those template parameters with the corresponding template arguments. This is done at the beginning of template argument deduction when any explicitly specified template arguments are substituted into the function type, and again at the end of template argument deduction when any template arguments that were deduced or obtained from default arguments are substituted.

7 The substitution occurs in all types and expressions that are used in the function type, in template parameter declarations, and in the template requirements (if any) (14.10.1). The expressions include not only constant expressions such as those that appear in array bounds or as nontype template arguments but also general expressions (i.e., non-constant expressions) inside `sizeof`, `decltype`, and other contexts that allow non-constant expressions. [*Note:* The equivalent substitution in exception specifications is done only when the function is instantiated, at which point a program is ill-formed if the substitution results in an invalid type or expression. — *end note*]

8 If a substitution results in an invalid type or expression or if a substituted template requirement cannot be satisfied (14.10.1.1), type deduction fails. An invalid type or expression is one that would be ill-formed if written using the substituted arguments. Access checking is not done as part of the substitution process. Consequently, when deduction succeeds, an access error could still result when the function is instantiated. Only invalid types and expressions in the immediate context of the function type and its template parameter types can result in a deduction failure. [*Note:* The evaluation of the substituted types and expressions can result in side effects such as the instantiation of class template specializations and/or function template specializations, the generation of implicitly-defined functions, etc. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

[*Example:*

```
struct X { };
struct Y {
  Y(X){}
};

template <class T> auto f(T t1, T t2) -> decltype(t1 + t2); // #1
X f(Y, Y);  // #2

X x1, x2;
X x3 = f(x1, x2);  // deduction fails on #1 (cannot add X+X), calls #2
```

— *end example*]

[*Note:* Type deduction may fail for the following reasons:

— Attempting to instantiate a pack expansion containing multiple parameter packs of differing lengths.

— Attempting to create an array with an element type that is `void`, a function type, a reference type, or an abstract class type, or attempting to create an array with a size that is zero or negative. [*Example:*

```
template <class T> int f(T[5]);
int I = f<int>(0);
int j = f<void>(0);             // invalid array
```

— *end example*]

— Attempting to use a type that is not a class type in a qualified name. [*Example:*

```
template <class T> int f(typename T::B*);
int i = f<int>(0);
```

— *end example*]

— Attempting to use a type in a *nested-name-specifier* of a *qualified-id* when that type does not contain the specified member, or

  — the specified member is not a type where a type is required, or

  — the specified member is not a template where a template is required, or

  — the specified member is not a non-type where a non-type is required.

[*Example:*

```
template <int I> struct X { };
template <template <class T> class> struct Z { };
template <class T> void f(typename T::Y*){}
template <class T> void g(X<T::N>*){}
template <class T> void h(Z<T::template TT>*){}
struct A {};
struct B { int Y; };
struct C {
  typedef int N;
};
struct D {
  typedef int TT;
};
```

```
int main() {
  // Deduction fails in each of these cases:
  f<A>(0);  // A does not contain a member Y
  f<B>(0);  // The Y member of B is not a type
  g<C>(0);  // The N member of C is not a non-type
  h<D>(0);  // The TT member of D is not a template
}
```

— *end example*]

— Attempting to create a pointer to reference type.

— Attempting to create a reference to `void`.

— Attempting to create "pointer to member of `T`" when `T` is not a class type. [*Example:*

```
template <class T> int f(int T::*);
int i = f<int>(0);
```

— *end example*]

— Attempting to give an invalid type to a non-type template parameter. [*Example:*

```
template <class T, T> struct S {};
template <class T> int f(S<T, T()>*);
struct X {};
int i0 = f<X>(0);
```

— *end example*]

— Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration. [*Example:*

```
template <class T, T*> int f(int);
int i2 = f<int,1>(0);           // can't conv 1 to int*
```

— *end example*]

— Attempting to create a function type in which a parameter has a type of `void`, or in which the return type is a function type or array type.

— Attempting to use a type in a *nested-name-specifier* of a *qualified-id* that refers to a member in a concept instance for which concept map lookup (14.10.1.1) does not find a concept map corresponding to that concept instance.

— Attempting to use a class or function template with template arguments that do not satisfy that template's requirements. [*Example:*

```
concept C<typename T> { /* ... */ }
template<typename T> requires C<T> class X { /* ... */ };

template<typename T> int f(X<T>*);  // #1
template<typename> int f(...);      // #2
int i0 = f<int>(0);                 // OK: calls #2
```

— *end example*]

— *end note*]

9 Except as described above, the use of an invalid value shall not cause type deduction to fail. [*Example:* In the following example 1000 is converted to `signed char` and results in an implementation-defined value as specified in (4.7). In other words, both templates are considered even though 1000, when converted to `signed char`, results in an implementation-defined value.

```
template <int> int f(int);
template <signed char> int f(int);
int i1 = f<1>(0);               // ambiguous
int i2 = f<1000>(0);            // ambiguous
```

— *end example*]

14.8.2.1 Deducing template arguments from a function call [temp.deduct.call]

1 Template argument deduction is done by comparing each function template parameter type (call it `P`) with the type of the corresponding argument of the call (call it `A`) as described below. If removing references and cv-qualifiers from `P` gives `std::initializer_list<`P'`>` for some P' and the argument is an initializer list (8.5.4), then deduction is performed instead for each element of the initializer list, taking P' as a function template parameter type and the initializer element as its argument. Otherwise, an initializer list argument causes the parameter to be considered a non-deduced context (14.8.2.5). [*Example:*

```
template<class T> void f(std::initializer_list<T>);
f({1,2,3});                 // T deduced to int
f({1,"asdf"});              // error: T deduced to both int and const char*

template<class T> void g(T);
g({1,2,3});                 // error: no argument deduced for T
```

— *end example*] For a function parameter pack, the type `A` of each remaining argument of the call is compared with the type `P` of the *declarator-id* of the function parameter pack. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. [*Note:* A function parameter pack can only occur at the end of a *parameter-declaration-list* (8.3.5). — *end note*] [*Example:*

```
template<class ... Types> void f(Types& ...);
template<class T1, class ... Types> void g(T1, Types ...);

void h(int x, float& y) {
  const int z = x;
  f(x, y, z);       // Types is deduced to int, float, const int
  g(x, y, z);       // T1 is deduced to int; Types is deduced to float, int
}
```

— *end example*]

2 If `P` is not a reference type:

— If `A` is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of `A` for type deduction; otherwise,

— If `A` is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of `A` for type deduction; otherwise,

— If `A` is a cv-qualified type, the top level cv-qualifiers of `A`'s type are ignored for type deduction.

3 If P is a cv-qualified type, the top level cv-qualifiers of P's type are ignored for type deduction. If P is a reference type, the type referred to by P is used for type deduction. If P is of the form T&&, where T is a template parameter, and the argument is an lvalue, the type A& is used in place of A for type deduction. [*Example:*

```
template <typename T> int f(T&&);
int i;
int j = f(i);                    // calls f<int&>(i)
template <typename T> int g(const T&&);
int k;
int n = g(k);                    // calls g<int>(k)
```

— *end example*]

4 In general, the deduction process attempts to find template argument values that will make the deduced A identical to A (after the type A is transformed as described above). However, there are three cases that allow a difference:

— If the original P is a reference type, the deduced A (i.e., the type referred to by the reference) can be more cv-qualified than the transformed A.

— The transformed A can be another pointer or pointer to member type that can be converted to the deduced A via a qualification conversion (4.4).

— If P is a class and P has the form *simple-template-id*, then the transformed A can be a derived class of the deduced A. Likewise, if P is a pointer to a class of the form *simple-template-id*, the transformed A can be a pointer to a derived class pointed to by the deduced A.

5 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails. [*Note:* if a *template-parameter* is not used in any of the function parameters of a function template, or is used only in a non-deduced context, its corresponding *template-argument* cannot be deduced from a function call and the *template-argument* must be explicitly specified. — *end note*]

6 When P is a function type, pointer to function type, or pointer to member function type:

— If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.

— If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set. If deduction succeeds for only one of the overload set members, that member is used as the argument value for the deduction. If deduction succeeds for more than one member of the overload set the parameter is treated as a non-deduced context.

7 [*Example:*

```
// Only one function of an overload set matches the call so the function
// parameter is a deduced context.
template <class T> int f(T (*p)(T));
int g(int);
int g(char);
int i = f(g);        // calls f(int (*)(int))
```

— *end example*]

8 [*Example:*

```
// Ambiguous deduction causes the second function parameter to be a
// non-deduced context.
template <class T> int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g);     // calls f(int, int (*)(int))
```

— *end example*]

9 [*Example:*

```
// The overload set contains a template, causing the second function
// parameter to be a non-deduced context.
template <class T> int f(T, T (*p)(T));
char g(char);
template <class T> T g(T);
int i = f(1, g);     // calls f(int, int (*)(int))
```

— *end example*]

14.8.2.2 Deducing template arguments taking the address of a function template [temp.deduct.funcaddr]

1 Template arguments can be deduced from the type specified when taking the address of an overloaded function (13.4). The function template's function type and the specified type are used as the types of P and A, and the deduction is done as described in 14.8.2.5.

14.8.2.3 Deducing conversion function template arguments [temp.deduct.conv]

1 Template argument deduction is done by comparing the return type of the conversion function template (call it P) with the type that is required as the result of the conversion (call it A) as described in 14.8.2.5.

2 If A is not a reference type:

— If P is an array type, the pointer type produced by the array-to-pointer standard conversion (4.2) is used in place of P for type deduction; otherwise,

— If P is a function type, the pointer type produced by the function-to-pointer standard conversion (4.3) is used in place of P for type deduction; otherwise,

— If P is a cv-qualified type, the top level cv-qualifiers of P's type are ignored for type deduction.

3 If A is a cv-qualified type, the top level cv-qualifiers of A's type are ignored for type deduction. If A is a reference type, the type referred to by A is used for type deduction. If P is a reference type, the type referred to by P is used for type deduction.

4 In general, the deduction process attempts to find template argument values that will make the deduced A identical to A. However, there are two cases that allow a difference:

— If the original A is a reference type, A can be more cv-qualified than the deduced A (i.e., the type referred to by the reference)

— The deduced A can be another pointer or pointer to member type that can be converted to A via a qualification conversion.

5 These alternatives are considered only if type deduction would otherwise fail. If they yield more than one possible deduced A, the type deduction fails.

6 When the deduction process requires a qualification conversion for a pointer or pointer to member type as described above, the following process is used to determine the deduced template argument values:

If A is a type

$cv_{1,0}$ "pointer to ..." $cv_{1,n-1}$ "pointer to" $cv_{1,n}$ *T1*

and P is a type

$cv_{2,0}$ "pointer to ..." $cv_{2,n-1}$ "pointer to" $cv_{2,n}$ *T2*

The cv-unqualified T1 and T2 are used as the types of A and P respectively for type deduction. [*Example:*

```
struct A {
  template <class T> operator T***();
};
A a;
const int * const * const * p1 = a;      // T is deduced as int, not const int
```

— *end example*]

14.8.2.4 Deducing template arguments during partial ordering [temp.deduct.partial]

1 Template argument deduction is done by comparing certain types associated with the two function templates being compared.

2 Two sets of types are used to determine the partial ordering. For each of the templates involved there is the original function type and the transformed function type. [*Note:* the creation of the transformed type is described in 14.5.6.2. — *end note*] The deduction process uses the transformed type as the argument template and the original type of the other template as the parameter template. This process is done twice for each type involved in the partial ordering comparison: once using the transformed template-1 as the argument template and template-2 as the parameter template and again using the transformed template-2 as the argument template and template-1 as the parameter template.

3 The types used to determine the ordering depend on the context in which the partial ordering is done:

— In the context of a function call, the function parameter types are used.

— In the context of a call to a conversion operator, the return types of the conversion function templates are used.

— In other contexts (14.5.6.2) the function template's function type is used.

4 Each type from the parameter template and the corresponding type from the argument template are used as the types of P and A.

5 Before the partial ordering is done, certain transformations are performed on the types used for partial ordering:

— If P is a reference type, P is replaced by the type referred to.

— If A is a reference type, A is replaced by the type referred to.

6 If both P and A were reference types (before being replaced with the type referred to above), determine which of the two types (if any) is more cv-qualified than the other; otherwise the types are considered to be equally cv-qualified for partial ordering purposes. The result of this determination will be used below.

7 Remove any top-level cv-qualifiers:

— If P is a cv-qualified type, P is replaced by the cv-unqualified version of P.

— If A is a cv-qualified type, A is replaced by the cv-unqualified version of A.

8 Using the resulting types P and A the deduction is then done as described in 14.8.2.5. If deduction succeeds for a given type, the type from the argument template is considered to be at least as specialized as the type from the parameter template.

9 If, for a given type, deduction succeeds in both directions (i.e., the types are identical after the transformations above) and if the type from the argument template is more cv-qualified than the type from the parameter template (as described above) that type is considered to be more specialized than the other. If neither type is more cv-qualified than the other then neither type is more specialized than the other.

10 If for each type being considered a given template is at least as specialized for all types and more specialized for some set of types and the other template is not more specialized for any types or is not at least as specialized for any types, then the given template is more specialized than the other template. Otherwise, neither template is more specialized than the other.

11 In most cases, all template parameters must have values in order for deduction to succeed, but for partial ordering purposes a template parameter may remain without a value provided it is not used in the types being used for partial ordering. [*Note:* a template parameter used in a non-deduced context is considered used. — *end note*] [*Example:*

```
template <class T> T f(int);        // #1
template <class T, class U> T f(U); // #2
void g() {
  f<int>(1);        // calls #1
}
```

— *end example*]

12 [*Note:* Partial ordering of function templates containing template parameter packs is independent of the number of deduced arguments for those template parameter packs. — *end note*] [*Example:*

```
template<class ...> struct Tuple { };
template<class ... Types> void g(Tuple<Types ...>);                 // #1
template<class T1, class ... Types> void g(Tuple<T1, Types ...>);   // #2
template<class T1, class ... Types> void g(Tuple<T1, Types& ...>);  // #3

g(Tuple<>());                   // calls #1
g(Tuple<int, float>());         // calls #2
g(Tuple<int, float&>());        // calls #3
g(Tuple<int>());                // calls #3
```

— *end example*]

14.8.2.5 Deducing template arguments from a type [temp.deduct.type]

1 Template arguments can be deduced in several different contexts, but in each case a type that is specified in terms of template parameters (call it P) is compared with an actual type (call it A), and an attempt is made to find template argument values (a type for a type parameter, a value for a non-type parameter, or a template for a template parameter) that will make P, after substitution of the deduced values (call it the deduced A), compatible with A.

2 In some cases, the deduction is done using a single set of types P and A, in other cases, there will be a set of corresponding types P and A. Type deduction is done independently for each P/A pair, and the deduced template argument values are then combined. If type deduction cannot be done for any P/A pair, or if for any pair the deduction leads to more than one possible set of deduced values, or if different pairs yield different

deduced values, or if any template argument remains neither deduced nor explicitly specified, template argument deduction fails.

3 A given type `P` can be composed from a number of other types, templates, and non-type values:

— A function type includes the types of each of the function parameters and the return type.

— A pointer to member type includes the type of the class object pointed to and the type of the member pointed to.

— A type that is a specialization of a class template (e.g., `A<int>`) includes the types, templates, and non-type values referenced by the template argument list of the specialization.

— An array type includes the array element type and the value of the array bound.

4 In most cases, the types, templates, and non-type values that are used to compose `P` participate in template argument deduction. That is, they may be used to determine the value of a template argument, and the value so determined must be consistent with the values determined elsewhere. In certain contexts, however, the value does not participate in type deduction, but instead uses the values of template arguments that were either deduced elsewhere or explicitly specified. If a template parameter is used only in non-deduced contexts and is not explicitly specified, template argument deduction fails.

5 The non-deduced contexts are:

— The *nested-name-specifier* of a type that was specified using a *qualified-id*.

— A non-type template argument or an array bound in which a subexpression references a template parameter.

— A template parameter used in the parameter type of a function parameter that has a default argument that is being used in the call for which argument deduction is being done.

— A function parameter for which argument deduction cannot be done because the associated function argument is a function, or a set of overloaded functions (13.4), and one or more of the following apply:

— more than one function matches the function parameter type (resulting in an ambiguous deduction), or

— no function matches the function parameter type, or

— the set of functions supplied as an argument contains one or more function templates.

— A function parameter for which the associated argument is an initializer list (8.5.4) but the parameter does not have `std::initializer_list` or reference to possibly cv-qualified `std::initializer_list` type. [*Example:*

```
template<class T> void g(T);
g({1,2,3});                 // error: no argument deduced for T
```

— *end example*]

6 When a type name is specified in a way that includes a non-deduced context, all of the types that comprise that type name are also non-deduced. However, a compound type can include both deduced and non-deduced types. [*Example:* If a type is specified as `A<T>::B<T2>`, both `T` and `T2` are non-deduced. Likewise, if a type is specified as `A<I+J>::X<T>`, `I`, `J`, and `T` are non-deduced. If a type is specified as `void f(typename A<T>::B, A<T>)`, the `T` in `A<T>::B` is non-deduced but the `T` in `A<T>` is deduced. — *end example*]

7 [*Example:* Here is an example in which different parameter/argument pairs produce inconsistent template argument deductions:

```
template<class T> void f(T x, T y) { /* ... */ }
struct A { /* ... */ };
struct B : A { /* ... */ };
void g(A a, B b) {
  f(a,b);           // error: T could be A or B
  f(b,a);           // error: T could be A or B
  f(a,a);           // OK: T is A
  f(b,b);           // OK: T is B
}
```

Here is an example where two template arguments are deduced from a single function parameter/argument pair. This can lead to conflicts that cause type deduction to fail:

```
template <class T, class U> void f(  T (*)( T, U, U )  );

int g1( int, float, float);
char g2( int, float, float);
int g3( int, char, float);

void r() {
  f(g1);            // OK: T is int and U is float
  f(g2);            // error: T could be char or int
  f(g3);            // error: U could be char or float
}
```

Here is an example where a qualification conversion applies between the argument type on the function call and the deduced template argument type:

```
template<class T> void f(const T*) { }
int *p;
void s() {
  f(p);             // f(const int*)
}
```

Here is an example where the template argument is used to instantiate a derived class type of the corresponding function parameter type:

```
template <class T> struct B { };
template <class T> struct D : public B<T> {};
struct D2 : public B<int> {};
template <class T> void f(B<T>&){}
void t() {
  D<int> d;
  D2     d2;
  f(d);             // calls f(B<int>&)
  f(d2);            // calls f(B<int>&)
}
```

— *end example*]

8 A template type argument T, a template template argument TT or a template non-type argument i can be deduced if P and A have one of the following forms:

```
T
cv-list T
T*
T&
```

```
T&&
T[integer-constant]
template-name<T>  (where template-name refers to a class template)
type(T)
T()
T(T)
T type::*
type T::*
T T::*
T (type::*)()
type (T::*)()
type (type::*)(T)
type (T::*)(T)
T (type::*)(T)
T (T::*)()
T (T::*)(T)
type[i]
template-name<i>  (where template-name refers to a class template)
TT<T>
TT<i>
TT<>
```

where `(T)` represents a *parameter-type-list* where at least one parameter type contains a `T`, and `()` represents a *parameter-type-list* where no parameter type contains a `T`. Similarly, `<T>` represents template argument lists where at least one argument contains a `T`, `<i>` represents template argument lists where at least one argument contains an `i` and `<>` represents template argument lists where no argument contains a `T` or an `i`.

9 If `P` has a form that contains `<T>` or `<i>`, then each argument $\texttt{P}_i$ of the respective template argument list `P` is compared with the corresponding argument $\texttt{A}_i$ of the corresponding template argument list of `A`. If the template argument list of `P` contains a pack expansion that is not the last template argument, the entire template argument list is a non-deduced context. If $\texttt{P}_i$ is a pack expansion, then the pattern of $\texttt{P}_i$ is compared with each remaining argument in the template argument list of `A`. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by $\texttt{P}_i$.

10 Similarly, if `P` has a form that contains `(T)`, then each parameter type $\texttt{P}_i$ of the respective *parameter-type-list* of `P` is compared with the corresponding parameter type $\texttt{A}_i$ of the corresponding *parameter-type-list* of `A`. If the *parameter-declaration* corresponding to $\texttt{P}_i$ is a function parameter pack, then the type of its *declarator-id* is compared with each remaining parameter type in the *parameter-type-list* of `A`. Each comparison deduces template arguments for subsequent positions in the template parameter packs expanded by the function parameter pack. [*Note:* A function parameter pack can only occur at the end of a *parameter-declaration-list* (8.3.5). — *end note*]

11 These forms can be used in the same way as `T` is for further composition of types. [*Example:*

```
X<int> (*)(char[6])
```

is of the form

```
template-name<T> (*)(type[i])
```

which is a variant of

```
type (*)(T)
```

where type is `X<int>` and `T` is `char[6]`. — *end example*]

12 Template arguments cannot be deduced from function arguments involving constructs other than the ones specified above.

13 A template type argument cannot be deduced from the type of a non-type *template-argument*.

14 [*Example:*

```
template<class T, T i> void f(double a[10][i]);
int v[10][20];
f(v);                // error: argument for template-parameter T cannot be deduced
```

— *end example*]

15 [*Note:* except for reference and pointer types, a major array bound is not part of a function parameter type and cannot be deduced from an argument:

```
template<int i> void f1(int a[10][i]);
template<int i> void f2(int a[i][20]);
template<int i> void f3(int (&a)[i][20]);

void g() {
  int v[10][20];
  f1(v);             // OK: i deduced to be 20
  f1<20>(v);         // OK
  f2(v);             // error: cannot deduce template-argument i
  f2<10>(v);         // OK
  f3(v);             // OK: i deduced to be 10
}
```

16 If, in the declaration of a function template with a non-type template parameter, the non-type template parameter is used in a subexpression in the function parameter list, the expression is a non-deduced context as specified above. [*Example:*

```
template <int i> class A { /* ... */ };
template <int i> void g(A<i+1>);
template <int i> void f(A<i>, A<i+1>);
void k() {
  A<1> a1;
  A<2> a2;
  g(a1);             // error: deduction fails for expression i+1
  g<0>(a1);          // OK
  f(a1, a2);         // OK
}
```

— *end example*] — *end note*] [*Note:* template parameters do not participate in template argument deduction if they are used only in non-deduced contexts. For example,

```
template<int i, typename T>
T deduce(typename A<T>::X x,     // T is not deduced here
  T t,                           // but T is deduced here
  typename B<i>::Y y);           // i is not deduced here
A<int> a;
B<77>  b;

int    x = deduce<77>(a.xm, 62, y.ym);
// T is deduced to be int, a.xm must be convertible to
// A<int>::X
```

```
// i is explicitly specified to be 77, y.ym must be convertible
// to B<77>::Y
```

— *end note*]

17 If, in the declaration of a function template with a non-type *template-parameter,* the non-type *template-parameter* is used in an expression in the function parameter-list and, if the corresponding *template-argument* is deduced, the *template-argument* type shall match the type of the *template-parameter* exactly, except that a *template-argument* deduced from an array bound may be of any integral type.[136] [*Example:*

```
template<int i> class A { /* ... */ };
template<short s> void f(A<s>);
void k1() {
  A<1> a;
  f(a);             // error: deduction fails for conversion from int to short
  f<1>(a);          // OK
}

template<const short cs> class B { };
template<short s> void g(B<s>);
void k2() {
  B<1> b;
  g(b);             // OK: cv-qualifiers are ignored on template parameter types
}
```

— *end example*]

18 A *template-argument* can be deduced from a function, pointer to function, or pointer to member function type.

[*Example:*

```
template<class T> void f(void(*)(T,int));
template<class T> void foo(T,int);
void g(int,int);
void g(char,int);

void h(int,int,int);
void h(char,int);
int m() {
  f(&g);            // error: ambiguous
  f(&h);            // OK: void h(char,int) is a unique match
  f(&foo);          // error: type deduction fails because foo is a template
}
```

— *end example*]

19 A template *type-parameter* cannot be deduced from the type of a function default argument. [*Example:*

```
template <class T> void f(T = 5, T = 7);
void g() {
  f(1);             // OK: call f<int>(1,7)
  f();              // error: cannot deduce T
  f<int>();         // OK: call f<int>(5,7)
}
```

136) Although the *template-argument* corresponding to a *template-parameter* of type `bool` may be deduced from an array bound, the resulting value will always be `true` because the array bound will be non-zero.

— *end example*]

20 The *template-argument* corresponding to a template *template-parameter* is deduced from the type of the *template-argument* of a class template specialization used in the argument list of a function call. [*Example:*

```
template <template <class T> class X> struct A { };
template <template <class T> class X> void f(A<X>) { }
template<class T> struct B { };
A<B> ab;
f(ab);              // calls f(A<B>)
```

— *end example*]

21 [*Note:* Template argument deduction involving parameter packs (14.5.3) can deduce zero or more arguments for each parameter pack. — *end note*] [*Example:*

```
template<class> struct X { };
template<class R, class ... ArgTypes> struct X<R(int, ArgTypes ...)> { };
template<class ... Types> struct Y { };
template<class T, class ... Types> struct Y<T, Types& ...> { };

template<class ... Types> int f(void (*)(Types ...));
void g(int, float);

X<int> x1;                      // uses primary template
X<int(int, float, double)> x2;  // uses partial specialization; ArgTypes contains float, double
X<int(float, int)> x3;          // uses primary template
Y<> y1;                         // use primary template; Types is empty
Y<int&, float&, double&> y2;    // uses partial specialization; T is int&, Types contains float, double
Y<int, float, double> y3;       // uses primary template; Types contains int, float, double
int fv = f(g);                  // OK; Types contains int, float
```

— *end example*]

22 If the original function parameter associated with A is a function parameter pack and the function parameter associated with P is not a function parameter pack, then template argument deduction fails. [*Example:*

```
template<class ... Args> void f(Args ... args);                    // #1
template<class T1, class ... Args> void f(T1 a1, Args ... args);   // #2
template<class T1, class T2> void f(T1 a1, T2 a2);                 // #3

f();                // calls #1
f(1, 2, 3);         // calls #2
f(1, 2);            // calls #3; non-variadic template #3 is more
                    // specialized than the variadic templates #1 and #2
```

— *end example*]

14.8.3 Overload resolution [temp.over]

1 A function template can be overloaded either by (non-template) functions of its name or by (other) function templates of the same name. When a call to that name is written (explicitly, or implicitly using the operator notation), template argument deduction (14.8.2) and checking of any explicit template arguments (14.3) are performed for each function template to find the template argument values (if any) that can be used with that function template to instantiate a function template specialization that can be invoked with the call

arguments. For each function template, if the argument deduction and checking succeeds, the *template-argument*s (deduced and/or explicit) are used to synthesize the declaration of a single function template specialization which is added to the candidate functions set to be used in overload resolution. If, for a given function template, argument deduction fails, no such function is added to the set of candidate functions for that template. The complete set of candidate functions includes all the synthesized declarations and all of the non-template overloaded functions of the same name. The synthesized declarations are treated like any other functions in the remainder of overload resolution, except as explicitly noted in 13.3.3.[137]

[*Example:*

```
template<class T> T max(T a, T b) { return a>b?a:b; }

void f(int a, int b, char c, char d) {
  int m1 = max(a,b);            // max(int a, int b)
  char m2 = max(c,d);           // max(char a, char b)
  int m3 = max(a,c);            // error: cannot generate max(int,char)
}
```

2 Adding the non-template function

```
int max(int,int);
```

to the example above would resolve the third call, by providing a function that could be called for `max(a,c)` after using the standard conversion of `char` to `int` for `c`.

3 Here is an example involving conversions on a function argument involved in *template-argument* deduction:

```
template<class T> struct B { /* ... */ };
template<class T> struct D : public B<T> { /* ... */ };
template<class T> void f(B<T>&);

void g(B<int>& bi, D<int>& di) {
  f(bi);            // f(bi)
  f(di);            // f((B<int>&)di)
}
```

4 Here is an example involving conversions on a function argument not involved in *template-parameter* deduction:

```
template<class T> void f(T*,int);       // #1
template<class T> void f(T,char);       // #2

void h(int* pi, int i, char c) {
  f(pi,i);          // #1: f<int>(pi,i)
  f(pi,c);          // #2: f<int*>(pi,c)

  f(i,c);           // #2: f<int>(i,c);
  f(i,i);           // #2: f<int>(i,char(i))
}
```

— *end example*]

137) The parameters of function template specializations contain no template parameter types. The set of conversions allowed on deduced arguments is limited, because the argument deduction process produces function templates with parameters that either match the call arguments exactly or differ only in ways that can be bridged by the allowed limited conversions. Non-deduced arguments allow the full range of conversions. Note also that 13.3.3 specifies that a non-template function will be given preference over a template specialization if the two functions are otherwise equally good candidates for an overload match.

5 Only the signature of a function template specialization is needed to enter the specialization in a set of candidate functions. Therefore only the function template declaration is needed to resolve a call for which a template specialization is a candidate. [*Example:*

```
template<class T> void f(T);      // declaration

void g() {
  f("Annemarie");                 // call of f<const char*>
}
```

6 The call of `f` is well-formed even if the template `f` is only declared and not defined at the point of the call. The program will be ill-formed unless a specialization for `f<const char*>`, either implicitly or explicitly generated, is present in some translation unit. — *end example*]

14.9 Concepts [concept]

1 Concepts describe an abstract interface that can be used to constrain templates (14.10). Concepts state certain syntactic and semantic requirements (14.9.1) on a set of template type, non-type, and template template parameters.

concept-id:
concept-name `<` *template-argument-list*$_{opt}$ `>`

concept-name:
identifier

2 A *concept-id* names a specific use of a concept by its *concept-name* and a set of template arguments. The concept and its template arguments, together, are referred to as a *concept instance*. [*Example:* `CopyConstructible<int>` is a *concept-id* if name lookup (3.4) determines that the identifier `CopyConstructible` refers to a *concept-name*; then, `CopyConstructible<int>` is a concept instance that refers to the `CopyConstructible` concept used with the type `int`. — *end example*]

3 A concept `C` is treated as a *constrained template* (14.10) `X` for the purpose of making the concept's definition a constrained context. The template requirements for `X` consist of a concept requirement `C<T1, T2, ..., TN>`, where `T1`, `T2`, ..., `TN` are the template parameters of `C`, and the template requirements implied by that concept requirement (14.10.1.2). [*Example:*

```
concept C<typename T> { }

concept D<typename T> {
  requires C<T>;
  // D is treated as a constrained template whose template requirements are D<T> && C<T>
}
```

— *end example*]

14.9.1 Concept definitions [concept.def]

1 The grammar for a *concept-definition* is:

concept-definition:
`auto`$_{opt}$ `concept` *identifier* `<` *template-parameter-list* `>`
refinement-clause$_{opt}$ *concept-body* `;`$_{opt}$

2 *Concept-definition* s are used to declare *concept-name* s. A *concept-name* is inserted into the scope in which it is declared immediately after the *concept-name* is seen. A concept is considered defined after the closing

brace of its *concept-body*. A *full concept name* is an identifier that is treated as if it were composed of the concept name and the sequence of its enclosing namespaces.

3 Concepts shall only be defined at namespace scope.

4 A *concept-definition* that starts with `auto` defines an ***`auto`*** *concept*.

5 The *template-parameter-list* of a *concept-definition* shall not contain any requirements specified in the simple form (14.10.1).

6

concept-body:
`{` *concept-member-specification*$_{opt}$ `}`

concept-member-specification:
concept-member-specifier concept-member-specification$_{opt}$

concept-member-specifier:
associated-function
type-parameter `;`
associated-requirements
axiom-definition

The body of a concept contains associated functions (14.9.1.1), associated types (14.9.1.2), associated class templates, associated requirements (14.9.1.3), and axioms (14.9.1.3).

14.9.1.1 Associated functions [concept.fct]

1 Associated functions describe functions, member functions, or operators (including templates thereof) that specify the functional behavior of the concept's template arguments and associated types and class templates (14.9.1.2). A concept map (14.9.2) for a given concept must satisfy each associated function in the concept (14.9.2.1).

associated-function:
simple-declaration
function-definition
template-declaration

2 An *associated-function* shall declare a function or function template. If the *declarator-id* of the declaration is a *qualified-id*, its *nested-name-specifier* shall name a template parameter of the enclosing concept; the declaration declares a member function or member function template. An associated function shall not be `extern`, inline or `virtual` (7.1.2), explicitly-defaulted or deleted (8.4), or a friend function (11.4). An associated function shall not contain an *exception-specification* (15.4).

3 Associated functions may specify requirements for non-member functions and operators. [*Example:*

```
concept Monoid<typename T> {
  T operator+(T, T);
  T identity();
}
```

— *end example*]

4 With the exception of the assignment operator (13.5.3) and operators `new`, `new[]`, `delete`, and `delete[]`, associated functions shall specify requirements for operators as non-member functions. [*Note:* This restriction applies even to the operators `()`, `[]`, and `->`, which can otherwise only be declared as non-static member functions (13.5): [*Example:*

```
concept Convertible<typename T, typename U> {
  operator U(T);               // OK: conversion from T to U
  T::operator U*() const;      // error: cannot specify requirement for member operator
}
```

— *end example*] — *end note*]

5 Associated functions may specify requirements for static or non-static member functions, constructors, and destructors. [*Example:*

```
concept Container<typename X> {
  X::X(int n);
  X::~X();
  bool X::empty() const;
  static size_t X::max_size();
}
```

— *end example*]

6 Associated functions may specify requirements for `new` and `delete`. [*Example:*

```
concept HeapAllocatable<typename T> {
  void* T::operator new(std::size_t);
  void* T::operator new[](std::size_t);
  void T::operator delete(void*);
  void T::operator delete[](void*);
}
```

— *end example*]

7 Associated functions may specify requirements for function templates and member function templates. [*Example:*

```
concept Sequence<typename X> {
  typename value_type;

  template<InputIterator Iter>
    requires Convertible<InputIterator<Iter>::value_type, Sequence<X>::value_type>
    X::X(Iter first, Iter last);
}
```

— *end example*]

8 Concepts may contain overloaded associated functions (clause 13). [*Example:*

```
concept C<typename X> {
  void f(X);
  void f(X, X);                 // OK
  int f(X, X);                  // error: differs only by return type
}
```

— *end example*]

9 Associated member functions with the same name and the same *parameter-type-list*, as well as associated member function templates with the same name, the same *parameter-type-list*, the same template parameter lists, and the same template requirements (if any), cannot be overloaded if any of them, but not all, have a *ref-qualifier* (8.3.5).

10 Associated functions may have a default implementation. This implementation is instantiated if used. A default implementation of an associated function is a constrained template (14.10) whose template requirements include concept requirements for the enclosing concept, its less refined concepts, and its associated requirements. [*Example:*

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
  bool operator!=(T x, T y) { return !(x == y); }
}

class X {};
bool operator==(const X&, const X&);

concept_map EqualityComparable<X> { } // OK, operator!= uses default
```

— *end example*]

14.9.1.2 Associated types and class templates [concept.assoc]

1 Associated types and associated class templates are types and class templates, respectively, defined in the concept body and used in the description of the concept.

2 An associated type specifies a type in a concept body. Associated types are typically used to express the parameter and return types of associated functions. [*Example:*

```
concept Callable1<typename F, typename T1> {
  typename result_type;
  result_type operator()(F&&, T1);
}
```

— *end example*]

3 An associated class template specifies a class template in a concept. [*Example:*

```
concept C<typename T> {
  template<ObjectType U> class X;
}
```

— *end example*]

4 Associated types and class templates may be provided with a default value. The default value is used to satisfy the associated type or class template requirement when no corresponding definition is provided in a concept map (14.9.2.2). [*Example:*

```
concept Iterator<typename Iter> {
  typename difference_type = int;
}

concept_map Iterator<int*> { } // OK, difference_type is int
```

— *end example*]

5 Associated types and class templates may use the simple form to specify requirements (14.10.1) on the associated type or class template. The simple form is equivalent to a declaration of the associated type or class template followed by an associated requirement (14.9.1.3) stated using the general form (14.10.1). [*Example:*

```
concept InputIterator<typename Iter> { /* ... */ }

concept Container<typename X> {
  InputIterator iterator; // same as typename iterator; requires InputIterator<iterator>;
}
```

— *end example*]

14.9.1.3 Associated requirements [concept.req]

1 Associated requirements place additional requirements on the concept's template parameters, associated types, and associated class templates. Associated requirements have the same form and behavior as template requirements in a constrained template (14.10).

associated-requirements:
requires-clause ;

[*Example:*

```
concept Iterator<typename Iter> {
  typename difference_type;
  requires SignedIntegral<difference_type>;
}
```

— *end example*]

14.9.1.4 Axioms [concept.axiom]

1 Axioms allow the expression of the semantic properties of concepts.

axiom-definition:
requires-clause$_{opt}$ `axiom` *identifier* `(` *parameter-declaration-clause* `)` *axiom-body*

axiom-body:
`{` *axiom-seq*$_{opt}$ `}`

axiom-seq:
axiom axiom-seq$_{opt}$

axiom:
expression-statement
`if (` *expression* `)` *expression-statement*

An *axiom-definition* defines a new semantic axiom whose name is specified by its *identifier*. [*Example:*

```
concept Semigroup<typename Op, typename T> : CopyConstructible<T> {
  T operator()(Op, T, T);

  axiom Associativity(Op op, T x, T y, T z) {
    op(x, op(y, z)) == op(op(x, y), z);
  }
}

concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
  T identity_element(Op);

  axiom Identity(Op op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
  }
}
```

— *end example*]

2 Within the body of an *axiom-definition*, equality (`==`) and inequality (`!=`) operators are available for each concept type parameter and associated type `T`. These implicitly-defined operators have the form:

```
bool operator==(const T&, const T&);
bool operator!=(const T&, const T&);
```

[*Example:*

```
concept CopyConstructible<typename T> {
  T::T(const T&);

  axiom CopyEquivalence(T x) {
    T(x) == x; // OK, uses implicit ==
  }
}
```

— *end example*]

3 Name lookup within an axiom will only find the implicitly-declared `==` and `!=` operators if the corresponding operation is not declared as an associated function (14.9.1.1) in the concept, one of its less refined concepts (14.9.3), or in an associated requirement (14.9.1.3). [*Example:*

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
  bool operator!=(T, T);

  axiom Reflexivity(T x) {
    x == x; // refers to EqualityComparable<T>::operator==
  }
}
```

— *end example*]

4 Where axioms state the equality of two expressions, implementations are permitted to replace one expression with the other. [*Example:*

```
template<typename Op, typename T> requires Monoid<Op, T>
  T identity(const Op& op, const T& t) {
    return op(t, identity_element(op)); // equivalent to return t;
  }
```

— *end example*]

5 Axioms can state conditional semantics using `if` statements. The *expression* is contextually converted to `bool` (clause 4). When the condition can be proven true, and the *expression-statement* states the equality of two expressions, implementations are permitted to replace one expression with the other. [*Example:*

```
concept TotalOrder<typename Op, typename T> {
  bool operator()(Op, T, T);

  axiom Antisymmetry(Op op, T x, T y) { if (op(x, y) && op(y, x)) x == y; }
  axiom Transitivity(Op op, T x, T y, T z) { if (op(x, y) && op(y, z)) op(x, z) == true; }
  axiom Totality(Op op, T x, T y) { (op(x, y) || op(y, x)) == true; }
}
```

— *end example*]

6 In a concept map where the requires clause of an *axiom-definition* is not satisfied, an implementation shall behave as if the axiom weere not defined. [*Example:*

```
concept EqualityComparable2<typename T, typename U = T> {
  bool operator==(T, U);
  bool operator!=(T, U);

  requires std::SameType<T, U> axiom Reflexivity(T x) {
      x == x; // OK: T and U have the same type
    }
}
```

— *end example*]

7 Whether an implementation replaces any expression according to an axiom is implementation-defined. With the exception of such substitutions, the presence of an axiom shall have no effect on the observable behavior of the program. [*Note:* the intent of axioms is to provide a mechanism to express the semantics of concepts. Such semantic information can be used for optimization, software verification, software testing, and other program analyses and transformations, all of which are outside the scope of this International Standard. — *end note*]

14.9.2 Concept maps [concept.map]

1 The grammar for a *concept-map-definition* is:

concept-map-definition:
`concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id* `{` *concept-map-member-specification*$_{opt}$ `}` `;`$_{opt}$

concept-map-member-specification:
concept-map-member concept-map-member-specification$_{opt}$

concept-map-member:
simple-declaration
function-definition
template-declaration

2 Concept maps describe how a set of template arguments satisfy the requirements stated in the body of a concept definition (14.9.1). For template argument deduction (14.8.2.5) against a constrained template to succeed, each of the template's requirements shall be satisfied (14.10.1.1). The concept map's name (which is the full concept name of its concept) is inserted into the scope in which the concept map or concept map template (14.5.8) is defined immediately after the *concept-id* is seen. [*Example:*

```
class student_record {
public:
  std::string id;
  std::string name;
  std::string address;
};

namespace N {
  concept EqualityComparable<typename T> {
    bool operator==(T, T);
  }
}

namespace M {
  concept_map N::EqualityComparable<student_record> { // the concept map's name is ::N::EqualityComparable
    bool operator==(const student_record& a, const student_record& b) {
      return a.id == b.id;
    }
```

```
  }
}

template<typename T> requires EqualityComparable<T> void f(T);

f(student_record());     // OK, have concept_map EqualityComparable<student_record>
```

— *end example*]

3 A concept map may contain two kinds of members: *requirement members* and members that satisfy requirement members. The latter may be explicitly declared within the concept map, explicitly declared within a concept map for a more refined concept, or generated implicitly from a default implementation from the concept or one of its more refined concepts.

4 Each requirement member represents an entity (a single associated function (14.9.1.1), associated type or associated class template (14.9.1.2) in the corresponding concept that must be satisfied as described below. The set of requirement members is the set of associated functions, associated types and associated class templates from the concept after substitution of the concept's template parameters with the corresponding template arguments. [*Note:* There is no way to explicitly declare a requirement member. — *end note*]

5 After a requirement is satisfied, the requirement member serves as a synonym for the set of entities that satisfies the requirement (14.9.2.1, 14.9.2.2). That set of entities is said to be the *satisfier* of the requirement member. Each requirement member is visible during qualified name lookup (3.4.3.3). [*Note:* A satisfier need not be a member of a concept map. — *end note*]

6 A concept map member that satisfies a requirement member cannot be found by any form of name lookup (3.4).

7 A concept map archetype (14.10.2) is considered to have satisfiers (generated from the concept) for each of its requirement members.

8 Concept maps shall satisfy every associated function (14.9.1.1), associated type and associated class template requirement (14.9.1.2) of its concept instance and all of the requirements inherited from its less refined concept instances (14.9.3). [*Example:*

```
concept C<typename T, typename U> { T f(T); U f(U); }

concept_map C<int, int> {
  int f(int); // OK: matches requirement for f in concept instance C<int, int>
}
```

— *end example*]

9 Concept maps shall not contain declarations that do not satisfy any requirement in their corresponding concept or its less refined concepts. [*Example:*

```
concept C<typename T> { }

concept_map C<int> {
  int f(int); // error: no requirement for function f
}
```

— *end example*]

10 At the point of definition of a concept map, all associated requirements (14.9.1.3) of the corresponding concept and its less refined concepts (14.9.3) shall be satisfied (14.10.1.1). [*Example:*

```
concept C<typename T> { /* ... */ }

concept D<typename Iter> {
  typename type;
  requires C<type>;
}

concept_map C<ptrdiff_t> { }

concept_map D<int*> {
  typedef ptrdiff_t type;
} // OK: there exists a concept map C<ptrdiff_t>

class X { ... };

concept_map D<X> {
  typedef long type;
} // error: no concept map C<long> if ptrdiff_t is not long
```

— *end example*]

11 A concept map for an `auto` concept is implicitly defined (14.9.1) when it is needed by concept map lookup (14.10.1.1). If any requirement of the concept or its less refined concepts would not be satisfied by the implicitly-defined concept map, the concept map is not implicitly defined. The implicitly-defined concept map is defined in the namespace of the concept. [*Example:*

```
auto concept C<typename T> {
  T::T(const T&);
  T operator+(T, T);
}

template<typename T>
requires C<T>
T add(T x, T y) {
  return x + y;
}

int f(int x, int y) {
  return add(x, y);      // OK: concept map C<int> implicitly defined
}
```

— *end example*]

12 [*Note:* Failure to implicitly define a concept map does not necessarily imply that the program is ill-formed (14.8.2). — *end note*] [*Example:*

```
auto concept F<typename T> {
  void f(T);
}

auto concept G<typename T> {
  void g(T);
}

template<typename T> requires F<T> void h(T); // #1
template<typename T> requires G<T> void h(T); // #2
```

```
struct X { };
void g(X);

void func(X x) {
  h(x);          // OK: implicit concept map F<X> fails, causing
                 // template argument deduction to fail for #1; calls #2
}
```

— *end example*]

13 A concept map or concept map template shall be defined before the first use of a concept instance that would make use of the concept map or concept map template in every translation unit in which such a use occurs; no diagnostic is required. If the introduction of a concept map or concept map template changes a previous result (e.g., in template argument deduction (14.8.2)), the program is ill-formed, no diagnostic required. Concept map templates must be instantiated if doing so would affect the semantics of the program. A concept map for a particular concept instance shall not be defined both implicitly and explicitly in the same namespace in a translation unit. If one translation unit of a program contains an explicitly-defined concept map for that concept instance, and a different translation contains an implicitly-defined concept map for that concept instance, then the program is ill-formed, no diagnostic required.

14 The implicit or explicit definition of a concept map asserts that the axioms (14.9.1.4) stated in its corresponding concept (and the concepts that it refines) hold, permitting an implementation to perform the transformations described in 14.9.1.4. If an axiom is violated, the behavior of the program is undefined.

14.9.2.1 Associated function definitions [concept.map.fct]

1 Function definitions in the concept map can be used to adapt the syntax of the concept's template arguments to the syntax expected by the concept. [*Example:*

```
concept Stack<typename S> {
  typename value_type;
  bool empty(S const&);
  void push(S&, value_type);
  void pop(S&);
  value_type& top(S&);

// Make a vector behave like a stack
template<Regular T>
concept_map Stack<std::vector<T> > {
  typedef T value_type;
  bool empty(std::vector<T> const& vec) { return vec.empty(); }
  void push(std::vector<T>& vec, value_type const& value) {
    vec.push_back(value);
  }
  void pop(std::vector<T>& vec) { vec.pop_back(); }
  value_type& top(std::vector<T>& vec) { return vec.back(); }
}
```

— *end example*]

2 A function or function template defined in a concept map is inline.

3 An associated function (or function template) requirement is satisfied as follows. Given an associated function (call it `f`), let `R` be the return type of `f`, after substitution of the concept's template arguments for their

corresponding concept parameters. Construct an expression `E` (as defined below) in the scope of the concept map. Then, the associated function requirement is satisfied:

— if `R` is *cv* `void` and the expression `E` is well-formed,

— otherwise, if `R` is not *cv* `void` and the expression "`E` implicitly converted to `R`" is well-formed, or

— otherwise, if `f` has a default implementation.

4 The expression `E` is defined differently depending on the associated function and the concept map definition. Let `parm1`, `parm2`, ..., `parm`N be the parameters of `f` (after substitution of the concept map arguments) and `parm1′`, `parm2′`, ..., `parm`N' be expressions, where each `parm`i' is an *id-expression* naming `parm`i. If the declared type of `parm`i is an lvalue reference type, then `parm`i' is treated as an lvalue, otherwise, `parm`i' is treated as an rvalue.

For an associated member function (or member function template) in a type `X` (after substitution of the concept map arguments into the associated member function or member function template), let `x` be an object of type *cv* `X`, where *cv* are the *cv-qualifiers* on the associated member function (or member function template). If the requirement has no *ref-qualifier* or if its *ref-qualifier* is `&`, `x` is an lvalue; otherwise, `x` is an rvalue.

The expression `E` is defined as follows:

— If `f` is an associated non-member function or function template and the concept map contains one or more function or function template definitions with the same name as `f`, `E` is `f(parm1′, parm2′, ..., parm`N'`)`, and the overload set of entities `f` consists of the definitions of `f` in the concept map. [*Note:* Unqualified lookup 3.4.1 and argument dependent lookup 3.4.2 are suppressed. — *end note*].

— Otherwise, if `f` is a non-static associated member function and the concept map contains one or more member function or member function template definitions in the type `X` and with the same name as `f`, `E` is `x.f(parm1′, parm2′, ..., parm`N'`)`, where name lookup of `x.f` refers to the definitions of `X::f` in the concept map.

— Otherwise, if `f` is a static associated member function and the concept map contains one or more member function or member function template definitions in the type `X` and with the same name as `f`, `E` is `X::f(parm1′, parm2′, ..., parm`N'`)`, where name lookup of `X::f` refers to the static definitions of `X::f` in the concept map.

— If the associated function or function template is a prefix unary operator `Op`, `E` is `Op parm1′`.

— If the associated function or function template is a postfix unary operator `Op`, `E` is `parm1′ Op`.

— If the associated function or function template is a binary operator `Op`, `E` is `parm1′ Op parm2′`.

— If the associated function or function template is the function call operator, `E` is `parm1′(parm2′, parm3′, ..., parm`N'`)`.

— If the associated function is a conversion operator, `E` is `parm1′` if the conversion operator requirement is not `explicit` and `(R)parm1′` if the conversion operator requirement is `explicit`, where `R` is the return type of the conversion operator.

— If the associated function or function template is a non-member function or function template, `E` is an unqualified call `f(parm1′, parm2′, ..., parm`N'`)`.

— If the associated function or function template is a static member function or function template in the type `X`, `E` is a call `X::f(parm1′, parm2′, ..., parm`N'`)`.

— If the associated function is a constructor or constructor template that is `explicit` or has $N \neq 1$ parameters, `E` is `X(parm1′, parm2′, ..., parm`N'`)`. [*Example:*

```
concept TwoIntConstructible<typename T> {
  T::T(int, int);
}

struct X { X(long, int); };
concept_map TwoIntConstructible<X> { }  // OK: X has a constructor that can accept two ints
                                        // (the first is converted to a long)
```

— *end example*]

— If the associated function is a constructor or constructor template that has one parameter (and is not `explicit`), `E` is "`parm1′` implicitly converted to `X`". [*Example:*

```
concept IC<typename T> {
  T::T(int);
}

concept EC<typename T> {
  explicit T::T(int);
}

struct X {
  X(int);
};

struct Y {
  explicit Y(int);
};

concept_map IC<X> { }   // OK
concept_map EC<X> { }   // OK
concept_map IC<Y> { }   // error: cannot copy-initialize Y from an int
concept_map EC<Y> { }   // OK
```

— *end example*]

— If the associated function is a destructor, `E` is `x.~X()`. [*Example:*

```
concept D<typename T> {
  T::~T();
}

concept_map D<int> { }  // OK: int is not a class type

struct X { };
concept_map D<X> { }    // OK: X has implicitly-declared, public destructor

struct Y { private: ~Y(); };
concept_map D<Y> { }    // error: Y's destructor is inaccessible
```

— *end example*]

— If the associated member function requirement is a requirement for an operator `new` or `new[]`, `E` is `operator new(parm1′, parm2′, ..., parm`N'`)` or `operator new[](parm1′, parm2′, ..., parm`N'`)`, respectively. If `X` is a class type, the allocation function's name is looked up in the scope of `X`. If this lookup fails to find the name, or if `X` is not a class type, the allocation function's name is looked up in the global scope.

— If the associated member function requirement is a requirement for an operator `delete` or `delete[]`, `E` is `operator delete(parm1′, parm2′, ..., parmN′)` or `operator delete[](parm1′, parm2′, ..., parmN′)`, respectively. If `X` is a class type, the deallocation function's name is looked up in the scope of `X`. If this lookup fails to find the name, or if `X` is not a class type, the deallocation function's name is looked up in the global scope.

— Otherwise, the associated function is a member function requirement, and `E` is `x.f(parm1′, parm2′, ..., parmN′)`.

5 Each satisfied associated function (or function template) requirement has a corresponding associated function candidate set. An *associated function candidate set* is a candidate set (14.10.3) representing the functions or operations used to satisfy the requirement. The seed of the associated function candidate set is determined based on the expression `E` used to determine that the requirement was satisfied.

— If the evaluation of `E` involves overload resolution at the top level, the seed is the function (13.3.1) selected by the outermost application of overload resolution (clause 13).

— Otherwise, if `E` is a pseudo destructor call (5.2.4), the seed is a *pseudo-destructor-name*.

— Otherwise, the seed is the initialization of an object.

14.9.2.2 Associated type and template definitions [concept.map.assoc]

1 Definitions in the concept map provide types and templates that satisfy requirements for associated types and templates (14.9.1.2), respectively.

2 Associated type parameter requirements are satisfied by type definitions in the body of a concept map. [*Example:*

```
concept ForwardIterator<typename Iter> {
  typename difference_type;
}

concept_map ForwardIterator<int*> {
  typedef ptrdiff_t difference_type;
}
```

— *end example*]

3 Associated class template requirements are satisfied by template aliases (14.5.7) in the body of the concept map. [*Example:*

```
concept Allocator<typename Alloc> {
  template<class T> class rebind;
}

template<typename T>
class my_allocator {
  template<typename U> class rebind;
};

template<typename T>
concept_map Allocator<my_allocator<T>> {
  template<class U> using rebind = my_allocator<T>::rebind;
}
```

— *end example*]

4 A concept map member that satisfies an associated type or class template requirement can be implicitly defined using template argument deduction (14.8.2) with one or more associated function requirements (14.9.2.1), if the associated type or class template requirement does not have a default value. The definition of the associated type or class template is determined using the rules of template argument deduction from a type (14.8.2.5).

— Let `P` be the return type of an associated function after substitution of the concept's template parameters specified by the concept map with their template arguments, and where each undefined associated type and associated class template has been replaced with a newly invented type or template template parameter, respectively.

— Let `A` be the return type of the seed in the associated function candidate set corresponding to the associated function.

If the deduction fails, no concept map members are implicitly defined by that associated function. If the results of deduction produced by different associated functions yield more than one possible value, that associated type or class template is not implicitly defined. [*Example:*

```
auto concept Dereferenceable<typename T> {
  typename value_type;
  value_type& operator*(T&);
}

template<typename T> requires Dereferenceable<T> void f(T&);

void g(int* x) {
  f(x); // OK: Dereferenceable<int*> implicitly defined
        // implicitly-defined Dereferenceable<int*>::operator* calls built-in * for integer pointers
        // implicitly-defined Dereferenceable<int*>::value_type is int
}
```

— *end example*]

5 If an associated type or class template (14.9.1.2) has a default value, a concept map member satisfying the associated type or class template requirement shall be implicitly defined by substituting the concept map arguments into the default value. If this substitution does not produce a valid type or template (14.8.2), the concept map member is not implicitly defined. [*Note:* If substitution fails, the associated type or class template can still be deduced, as described below. — *end note*] [*Example:*

```
auto concept A<typename T> {
  typename result_type = typename T::result_type;
}

auto concept B<typename T> {
  T::T(const T&);
}

template<typename T> requires A<T> void f(const T&); // #1
template<typename T> requires B<T> void f(const T&); // #2

struct X {};
void g(X x) {
  f(x); // OK: A<X> cannot satisfy result_type requirement, and is not implicitly defined, calls #2
}
```

— *end example*]

14.9.3 Concept refinement [concept.refine]

1 The grammar for a *refinement-clause* is:

refinement-clause:
: *refinement-specifier-list*

refinement-specifier-list:
refinement-specifier , *refinement-specifier-list*
refinement-specifier

refinement-specifier:
concept-instance-alias-def$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

concept-instance-alias-def:
identifier =

2 Refinements specify an inheritance relationship among concepts. A concept `B` named in a *refinement-specifier* of concept `D` is a *less refined concept* of `D` and `D` is a *more refined concept* of `B`. A concept refinement inherits all requirements in the body of a concept (14.9.1), such that the requirements of the more refined concept are a superset of the requirements of the less refined concept. [*Note:* when a concept `D` refines a concept `B`, every set of template arguments that satisfies the requirements of `D` also satisfies the requirements of `B`. The refinement relationship is transitive. — *end note*] [*Example:* In the following example, `EquilateralPolygon` refines `Polygon`. Thus, every `EquilateralPolygon` is a `Polygon`, and constrained templates (14.10) that are well-formed with a `Polygon` constraint are well-formed when given an `EquilateralPolygon`.

```
concept Polygon<typename P> { /* ... */ }

concept EquilateralPolygon<typename P> : Polygon<P> { /* ... */ }
```

— *end example*]

3 A *refinement-specifier* shall refer to a previously defined concept. [*Example:*

```
concept C<typename T> : C<vector<T>> {/* ... */ } // error: concept C is not defined
```

— *end example*]

4 The *template-argument-list* of a *refinement-specifier*'s *concept-id* shall refer to at least one of the template parameters, and no template parameter shall be used in a way that establishes an archetype (14.10.2). [*Example:*

```
concept C<typename T> { T f(); }

concept D<typename T>
  : C<int>  // error: C<int> uses no template parameters
{
  // ...
}

concept E<typename T>
  : C<T>, D<decltype(f())> // error: establishes archetype T′
{
  // ...
}
```

— *end example*]

5 Within the definition of a concept, a concept map archetype is synthesized for each *refinement-specifier* in the concept's *refinement-clause* (if any).

14.9.3.1 Concept member lookup [concept.member.lookup]

1 Concept member lookup determines the meaning of a name in concept scope (3.3.7). The following steps define the result of name lookup for a member name `f` in concept scope `C`. C_R is the set of concept scopes corresponding to the concepts refined by the concept whose scope is `C`.

2 If the name `f` is declared in concept scope `C`, and `f` refers to an associated type or class template (14.9.1.2), then the result of name lookup is the associated type or class template.

3 If the name `f` is declared in concept scope `C`, and `f` refers to one or more associated functions (14.9.1.1), then the result of name lookup is the set consisting of the associated functions in `C` in addition to the associated functions in each concept scope in C_R for which name lookup of `f` results in a set of associated functions. [*Example:*

```
concept C1<typename T> : CopyConstructible<T> {
  T f(T); // #1
}

concept C2<typename T> {
  typename f;
}

concept D<typename T> : C1<T>, C2<T> {
  T f(T, T); // #2
}

template<typename T>
requires D<T>
void f(T x)
{
  D<T>::f(x); // name lookup finds #1 and #2, overload resolution selects #1
}
```

— *end example*]

4 If the name `f` is not declared in `C`, name lookup searches for `f` in the scopes of each of the less refined concepts (C_R). If name lookup of `f` is ambiguous in any concept scope C_R, name lookup of `f` in `C` is ambiguous. Otherwise, the set of concept scopes $C_{R'}$ is a subset of C_R containing only those concept scopes for which name lookup finds `f`. The result of name lookup for `f` in `C` is defined by:

- If $C_{R'}$ is empty, name lookup of `f` in `C` returns no result.
- Otherwise, if $C_{R'}$ contains only a single concept scope, name lookup for `f` in `C` is the result of name lookup for `f` in that concept scope.
- Otherwise, if `f` refers to one or more functions in all of the concept scopes in $C_{R'}$, then `f` refers to the set consisting of all associated functions from all of the concept scopes in $C_{R'}$.
- Otherwise, if `f` refers to an associated type or class template in all concept scopes in $C_{R'}$, and all of the associated types or class templates are equivalent (14.10.1), the result is the associated type or class template `f`.
- Otherwise, name lookup of `f` in `C` is ambiguous.

[*Example:*

```
concept A<typename T> { typename t; }

concept B<typename T> { typename t; }

concept C<typename T> :  A<T>, B<T> {
  f(t);                  // error: ambiguous, the two t's are not equivalent
  f(A<T>::t);            // OK
}
```

— *end example*]

5 When name lookup in a concept scope `C` results in a set of associated functions, duplicate associated functions are removed from the set. [*Example:*

```
concept A<typename T> {
  T f(T); // #1a
}

concept B<typename T> {
  T f(T); // #1b
  T g(T); // #2a
}

concept C<typename T> : A<T>, B<T> {
  T g(T); // #2b
}

template<typename T>
requires C<T>
void h(T x) {
  C<T>::f(x); // overload set contains #1a; #1b was removed as a duplicate
  C<T>::g(x); // overload set contains #2b; #2a was removed as a duplicate
}
```

— *end example*]

14.9.3.2 Implicit concept maps for refined concepts [concept.refine.maps]

1 When a concept map or concept map template is defined for a concept `C` that has a refinement clause, concept maps or concept map templates for each of the concept instances in the refinement clause of `C` shall be defined, implicitly or explicitly, in the namespace of which the more refined concept map or concept map template is a member. [*Example:*

```
concept A<typename T> { }
concept B<typename T> : A<T> { }

concept_map B<int> { } // implicitly defines concept map A<int>
```

— *end example*]

2 A concept map or concept map template for a refinement (called a *less refined* concept map or concept map template) has been defined if concept map lookup (14.10.1.1) finds a concept map for the concept instance `I` in the namespace of which the more refined concept map or concept map template is a member, where `I` is determined by substituting the template arguments of the more refined concept map or concept map template into the *refinement-specifier* corresponding to the refinement. If concept map lookup fails, if it

finds an implicitly generated concept map for an `auto` concept, or if it finds a concept map from a different namespace, a suitable concept map is defined implicitly, as described below. [*Example:*

```
concept C<typename T> { }
concept D<typename T> : C<T> { }

template<typename T> concept_map C<T*> { } // #1

template<typename T>
concept_map D<T*> { }   // OK: #1 defines the concept map corresponding to the refinement-specifier C<T*>
```

— *end example*]

3 Concept map templates (14.5.8) can only be implicitly defined for certain less refined concepts from the concept map templates of more refined concepts. A less refined concept map template corresponding to a particular *refinement-specifier* can be defined if all of the template parameters of the more refined concept map template can be deduced. Let `R` be the *refinement-specifier* after substitution of the more refined concept map's template arguments for the corresponding template parameters. A template parameter `T` of the more refined concept map template can be deduced if for any template argument `P` in `R`'s *template-argument-list* there exists a type, template, or value `A` such that template argument deduction (14.8.2) performed with that `P`/`A` pair would determine a value for `T`. If any template parameter of the more refined concept map template can not be deduced, the program is ill-formed. Otherwise, a less refined concept map template is implicitly defined in the namespace of the more refined concept map template; its *template-parameter-list* and template requirements are the same as the more refined concept map template, and its *concept-id* is `R`. [*Example:*

```
concept C<typename T> { }
concept D<typename T, typename U> : C<T> { }

template<typename T> struct A { };

template<typename T> concept_map D<A<T>, T> { }
// implicitly defines:
template<typename T> concept_map C<A<T>> { }

template<typename T, typename U>
  concept_map D<T, A<U>> { } // ill-formed: cannot deduce template parameter U from C<T>
                             // and there is no concept map template C<T>
```

— *end example*]

4 When a less refined concept map or concept map template is implicitly defined, definitions in the more refined concept map or concept map template can be used to satisfy the requirements of the less refined concept (14.9.2). [*Note:* a single function definition in a concept map can be used to satisfy multiple requirements. — *end note*] [*Example:* in this example, the concept map `D<int>` implicitly defines the concept map `C<int>`.

```
concept C<typename T> {
  T f(T);
  void g(T);
}

concept D<typename T> : C<T> {
  void g(T);
}
```

```
concept_map D<int> {
  int f(int x) { return -x; } // satisfies requirement for C<int>::f
  void g(int x) { } // satisfies requirement for C<int>::g and D<int>::g
}
```

— *end example*]

5 Each concept map or concept map template shall have satisfiers that are compatible with the satisfiers of its less refined concept maps or concept map templates. A satisfier of the more refined concept map or concept map template is compatible with its corresponding satisfier of the less refined concept map or concept map template if

— the satisfiers correspond to an associated function requirement (14.9.1.1) and their associated function candidate sets have the same seed or

— the satisfiers satisfy an associated type or class template requirement (14.9.1.2) and both satisfiers name the same type or template, respectively.

If a program contains satisfiers of a concept map or concept map template that are not compatible with their corresponding satisfiers of a less refined concept map or concept map template, the program is ill-formed. If the concept maps or concept map templates with satisfiers that are not compatible occur in different translation units, no diagnostic is required. [*Example:*

```
concept C<typename T> {
  typename assoc;
  assoc f(T);
}

concept D<typename T> : C<T> {
  int g(T);
}

concept E<typename T> : D<T> { }

concept_map C<int> {
  typedef int assoc;
  int f(int x) { return x; }
}

concept_map D<int> {
  typedef int assoc;            // OK: same type as C<int>::assoc
                                // OK: f is not defined in D<int>
  int g(int x) { return -x; }   // OK: satisfies D<int>::g
}

concept_map E<int> {
  typedef float assoc;          // error: E<int>::assoc and D<int>::assoc are not the same type
                                // OK: f is not defined in D<int>
  int g(int x) { return x; }    // error: D<int>::g already defined in concept map D<int>
}
```

— *end example*]

14.9.4 Support concepts [concept.support]

1 The concepts in [concept.support] provide the ability to state template requirements for C++ type classi-

fications (3.9) and type relationships that cannot be expressed directly with concepts. Concept maps for these concepts are implicitly defined. A program shall not provide a concept map or concept map template for any concept in [concept.support] nor shall it provide a definition for any of these concepts.

2 The following concepts are implicitly defined at the beginning of each translation unit. [*Note:* This implies that the namespace `std` is always visible at global scope. — *end note*]

```
namespace std {
  concept Returnable<typename T> { }
  concept PointeeType<typename T> { }
  concept MemberPointeeType<typename T> { }
  concept ReferentType<typename T> { }
  concept VariableType<typename T> { }
  concept ObjectType<typename T> see below;
  concept ValueType<typename T> see below;
  concept ClassType<typename T> see below;
  concept Class<typename T> see below;
  concept PolymorphicClass<typename T> see below;
  concept Union<typename T> see below;
  concept TrivialType<typename T> see below;
  concept StandardLayoutType<typename T> see below;
  concept LiteralType<typename T> see below;
  concept ScalarType<typename T> see below;
  concept ArithmeticType<typename T> see below;
  concept NonTypeTemplateParameterType<typename T> see below;
  concept IntegralConstantExpressionType<typename T> see below;
  concept IntegralType<typename T> see below;
  concept EnumerationType<typename T> see below;
  concept FloatingPointType<typename T> see below;
  concept SameType<typename T, typename U> {  }
  concept DerivedFrom<typename Derived, typename Base> { }
}
```

`concept Returnable<typename T> { }`

3 *Note:* Describes types that can be used as the return type of a function.

4 *Requires:* for every non-array type `T` that is *cv* `void` or that provides a copy operation suitable for use in a return statement (6.6.3) and is not an abstract class, the concept map `Returnable<T>` is implicitly defined in namespace `std`.

`concept PointeeType<typename T> { }`

5 *Note:* describes types to which a pointer can be created.

6 *Requires:* for every type `T` that is an object type, a function type that does not have cv-qualifiers, or *cv* `void`, a concept map `PointeeType<T>` is implicitly defined in namespace `std`.

`concept MemberPointeeType<typename T> { }`

7 *Note:* describes types to which a pointer-to-member can be created.

8 *Requires:* for every type `T` that is an object type or function type, a concept map `MemberPointeeType<T>` is implicitly defined in namespace `std`.

`concept ReferentType<typename T> { }`

9 *Note:* describes types to which a reference can be created, including reference types (since references to references can be formed during substitution of template arguments).

10 *Requires:* for every type `T` that is an object type, a function type that does not have cv-qualifiers, or a reference type, a concept map `ReferentType<T>` is implicitly defined in namespace `std`.

```
concept VariableType<typename T> : ReferentType<T> { }
```

11 *Note:* describes types that can be used to declare a variable.

12 *Requires:* for every type `T` that is an object type or reference type, but not an abstract class, a concept map `VariableType<T>` is implicitly defined in namespace `std`.

```
concept ObjectType<typename T> : ReferentType<T>, PointeeType<T> { }
```

13 *Note:* describes object types (3.9), for which storage can be allocated.

14 *Requires:* for every type `T` that is an object type, a concept map `ObjectType<T>` is implicitly defined in namespace `std`.

```
concept ValueType<typename T> : ObjectType<T>, VariableType<T>, MemberPointeeType<T> { }
```

15 *Note:* describes value types, for which objects, variables, references, pointers, and pointers-to-members can be created.

16 *Requires:* for every type `T` that is an object type but not an abstract class, a concept map `ValueType<T>` is implicitly defined in namespace `std`.

```
concept ClassType<typename T> : ObjectType<T> { }
```

17 *Note:* describes class types (i.e., unions, classes, and structs).

18 *Requires:* for every type `T` that is a class type (Clause 9), a concept map `ClassType<T>` is implicitly defined in namespace `std`.

```
concept Class<typename T> : ClassType<T> { }
```

19 *Note:* describes non-union classes (Clause 9).

20 *Requires:* for every non-union class `T`, a concept map `Class<T>` is implicitly defined in namespace `std`.

```
concept PolymorphicClass<typename T> : Class<T> { }
```

21 *Note:* describes polymorphic class types (10.3).

22 *Requires:* for every type `T` that is a polymorphic class, a concept map `PolymorphicClass<T>` is implicitly defined in namespace `std`.

```
concept Union<typename T> : ClassType<T>, ValueType<T> { }
```

23 *Note:* describes union types (9.5).

24 *Requires:* for every type `T` that is a union, a concept map `Union<T>` is implicitly defined in namespace `std`.

```
concept TrivialType<typename T> : ValueType<T> { }
```

25 *Note:* describes trivial types (3.9).

26 *Requires:* for every type `T` that is a trivial type, a concept map `TrivialType<T>` is implicitly defined in namespace `std`.

```
concept StandardLayoutType<typename T> : ValueType<T> { }
```

27 *Note:* describes standard-layout types (3.9).

28 *Requires:* for every type `T` that is a standard-layout type, a concept map `StandardLayoutType<T>` is implicitly defined in namespace `std`.

```
concept LiteralType<typename T> : ValueType<T> { }
```

29 *Note:* describes literal types (3.9).

30 *Requires:* for every type `T` that is a literal type, a concept map `LiteralType<T>` is implicitly defined in namespace `std`.

```
concept ScalarType<typename T>
  : TrivialType<T>, LiteralType<T>, StandardLayoutType<T> { }
```

31 *Note:* describes scalar types (3.9).

32 *Requires:* for every type `T` that is a scalar type, a concept map `ScalarType<T>` is implicitly defined in namespace `std`.

```
concept ArithmeticType<typename T> : ScalarType<T> { }
```

33 *Note:* describes arithmetic types (3.9.1).

34 *Requires:* for every type `T` that is an arithmetic type, a concept map `ArithmeticType<T>` is implicitly defined in namespace `std`.

```
concept NonTypeTemplateParameterType<typename T> : VariableType<T> { }
```

35 *Note:* describes type that can be used as the type of a non-type template parameter (14.1).

36 *Requires:* for every type `T` that can be the type of a non-type *template-parameter* (14.1), a concept map `NonTypeTemplateParameterType<T>` is implicitly defined in namespace `std`.

```
concept IntegralConstantExpressionType<typename T>
  : ScalarType<T>, NonTypeTemplateParameterType<T> { }
```

37 *Note:* describes types that can be the type of an integral constant expression (5.19).

38 *Requires:* for every type `T` that is an integral type or enumeration type, a concept map `IntegralConstantExpressionTyp` is implicitly defined in namespace `std`.

```
concept IntegralType<typename T>
  : IntegralConstantExpressionType<T>, ArithmeticType<T> { }
```

39 *Note:* describes integral types (3.9.1).

40 *Requires:* for every type `T` that is an integral type, a concept map `IntegralType<T>` is implicitly defined in namespace `std`.

```
concept EnumerationType<typename T> : IntegralConstantExpressionType<T> {
  IntegralType underlying_type;
}
```

41 *Note:* describes enumeration types (7.2). `underlying_type` is the underlying type of the enumeration type.

42 *Requires:* for every type `T` that is an enumeration type, a concept map `EnumerationType<T>` is implicitly defined in namespace `std`.

```
concept FloatingPointType<typename T> : ArithmeticType<T> { }
```

43 *Note:* describes floating point types (3.9.1).

44 *Requires:* for every type `T` that is a floating point type, a concept map `FloatingPointType<T>` is implicitly defined in namespace `std`.

```
concept SameType<typename T, typename U> { }
```

45 *Note:* describes a same-type requirement (14.10.1).

```
concept DerivedFrom<typename Derived, typename Base> { }
```

46 *Requires:* for every pair of class types (`T`, `U`), such that `T` is either the same as or publicly and unambiguously derived from `U`, a concept map `DerivedFrom<T, U>` is implicitly defined in namespace `std`.

14.10 Constrained templates [temp.constrained]

1 A template that has a *requires-clause* (or declares any template type parameters using the simple form of requirements (14.1)) is a *constrained template*. A constrained template can only be instantiated with template arguments that satisfy its template requirements. The template definitions of constrained templates are similarly constrained, requiring names to be found through name lookup at template definition time (3.4). [*Note:* Names can be found in the template requirements of a constrained template (3.3.8). The practical effect of constrained templates is that they provide improved diagnostics at template definition time, such that any use of the constrained template that satisfies the template's requirements is likely to result in a well-formed instantiation. — *end note*]

2 A template that is not a constrained template is an *unconstrained template*.

3 A *constrained context* is a part of a constrained template in which all name lookup is resolved at template definition time. Names that would be dependent outside of a constrained context shall be found in the current scope, which includes the template requirements of the constrained template (3.3.8). [*Note:* Within a constrained context, template parameters behave as if aliased to their corresponding archetypes (14.10.2) so there are no dependent types (14.6.2.1), and no type-dependent values (14.6.2.2) or dependent names (14.6.2). Instantiation in constrained contexts (14.10.4) still substitutes types, templates and values for template parameters. — *end note*] A constrained context is any part of a constrained template that is not an unconstrained context (described below).

4 Any context that is not a constrained context is an *unconstrained context*. Within a constrained context, several constructs provide unconstrained contexts:

- a late-checked block (6.9),
- a default argument in a *template-parameter*, and
- a default function argument (8.3.5).

5 Within a constrained context, a program shall not require a template specialization of an unconstrained template for which the template arguments of the specialization depend on a template parameter.

14.10.1 Template requirements [temp.req]

1 A template has *template requirements* if it contains a *requires-clause* or any of its template parameters is specified using the simple form of requirements (14.1). Template requirements state the conditions under which the template can be used.

requires-clause:
requires *requirement-list*
requires **(** *requirement-list* **)**

requirement-list:
requirement **...**$_{opt}$ **&&** *requirement-list*
requirement **...**$_{opt}$

requirement:
concept-instance-alias-def$_{opt}$ **::**$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
! **::**$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

2 A *requires-clause* contains a list of requirements, all of which must be satisfied by the template arguments for the template. [*Note:* Requirement satisfaction is described in 14.10.1.1. — *end note*] A *requirement* not containing a `!` is a *concept requirement*. A *requirement* containing a `!` is a *negative requirement*.

3 A concept requirement that refers to the `std::SameType` concept (14.9.4) is a *same-type requirement*. A same-type requirement is satisfied when its two template arguments refer to the same type (including the same *cv* qualifiers). In a constrained template (14.10), a same-type requirement `std::SameType<T1, T2>` makes the types `T1` and `T2` equivalent. [*Note:* type equivalence is a congruence relation, thus

— `std::SameType<T1, T2>` implies `std::SameType<T2, T1>`,

— `std::SameType<T1, T2>` and `std::SameType<T2, T3>` implies `std::SameType<T1, T3>`,

— `std::SameType<T1, T1>` is trivially true,

— `std::SameType<T1*, T2*>` implies `std::SameType<T1, T2>` and `std::SameType<T1**, T2**>`, etc.

— *end note*] [*Example:*

```
concept C<typename T> {
  typename assoc;
  assoc a(T);
}

concept D<typename T> {
  T::T(const T&);
  T operator+(T, T);
}

template<typename T, typename U>
requires C<T> && C<U> && std::SameType<C<T>::assoc, C<U>::assoc> && D<C<T>::assoc>
C<T>::assoc f(T t, U u) {
  return a(t) + a(u); // OK: C<T>::assoc and C<U>::assoc are the same type
}
```

— *end example*]

4 A *requirement* followed by an ellipsis is a pack expansion (14.5.3). Requirement pack expansions place requirements on all of the arguments in one or more template parameter packs. [*Example:*

```
auto concept OutputStreamable<typename T> {
  std::ostream& operator<<(std::ostream&, const T&);
}

template<typename T, typename... Rest>
requires OutputStreamable<T> && OutputStreamable<Rest>...
void print(const T& t, const Rest&... rest) {
  std::cout << t;
```

```
  print(rest...);
}

template<typename T>
requires OutputStreamable<T>
void print(const T& t) {
  std::cout << t;
}

void f(int x, float y) {
  print(17, ", ", 3.14159);      // OK: implicitly-generated OutputStreamable<int>,
                                 // OutputStreamable<const char[3]>,
                                 // and OutputStreamable<double>
  print(17, " ", std::cout);     // error: no concept map OutputStreamable<std::ostream>
}
```

— *end example*]

5 If the requirements of a template are such that no set of template arguments can satisfy all of the requirements, the program is ill-formed, no diagnostic required. [*Example:*

```
concept C<typename T> { }

template<typename T>
requires C<T> && !C<T>
void f(const T&);        // error: no type can satisfy both C<T> && !C<T>, no diagnostic required
```

— *end example*]

6 A *concept-instance-alias-def* defines its *identifier* to be an alias of the concept instance given in its *requirement* or *refinement-specifier* . When the *concept-instance-alias-def* appears in a *member-requirement* (9.2), the potential scope of the *identifier* begins at its point of declaration and terminates at the end of the constrained member's declaration. When the *concept-instance-alias-def* appears in the optional *requires-clause* of an *axiom-definition* (14.9.1.4), the potential scope of the *identifier* begins at its point of declaration and terminates at the end of the *axiom-definition*. Otherwise, a *concept-instance-alias-def* inserts the *identifier* as a name in the scope of:

- the template parameters of the concept, when the *concept-instance-alias-def* appears in a *refinement-specifier* (14.9.3);
- the enclosing concept, when the *concept-instance-alias-def* appears in the *associated-requirements* (14.9.1.3); or
- the template parameters declared in the *template-parameter-list* immediately before the `requires` keyword, when the *concept-instance-alias-def* appears in the optional *requires-clause* of a *template-declaration*.

7 [*Example:*

```
concept A<typename X, typename Y, typename Z> {
  typename result_type;
}

concept B<typename X, typename Y> {
  typename result_type;
}
```

```
concept C<typename X> {
  typename R;
}

template <class T>
  requires J = C<T>
  J::R f(T);          // qualified lookup finds type name R
                      // within the concept C (3.4.3.3)

auto concept D<typename Op, typename Elem> {
  requires a = A<Op, Elem, Elem>;
  requires B<a::result_type, Elem>;
  typename result_type = a::result_type;
}
```

— *end example*]

8 If a *concept-instance-alias-def* appears in a *requirement* that is the pattern of a pack expansion, the program is ill-formed. [*Example:*

```
concept C<typename... Args> { }

template <class... Args>
  requires a = C<Args>...      // error: requirement aliases may refer only
                               // to requirements that are not pack expansions

  void f(Args...);
```

— *end example*]

14.10.1.1 Requirement satisfaction [temp.req.sat]

1 During template argument deduction (14.8.2) against a constrained template, it is necessary to determine whether each of the requirements of the constrained template can be satisfied by the template arguments.

2 A concept requirement is *satisfied* if concept map lookup (described below) finds a unique concept map with the same full concept name as the concept named by the concept requirement and whose template argument list is the same as the template argument list of the concept requirement, after substitution of the constrained template's template arguments into the concept requirement's template argument list. Concept maps used to satisfy a concept requirement can be defined explicitly (14.9.2), instantiated from a concept map template (14.5.8), or defined implicitly (14.9.2). [*Example:*

```
concept A<typename T> { }
auto concept B<typename T> { T operator+(T, T); }
concept C<typename T> { }
concept D<typename T> { }

concept_map A<float> { }
concept_map B<float> { }
template<typename T> concept_map C<T*> { }
template<typename T> requires B<T> concept_map D<T> { }

template<typename T> requires A<T> void f(T);
template<typename T> requires B<T> void g(T);
template<typename T> requires C<T> void h(T);
template<typename T> requires D<T> void i(T);
```

```
struct X { };
void h(float x, int y, int X::* p, int *q) {
  f(x);       // OK: uses concept map A<float>
  f(y);       // error: no concept map A<int>; requirement not satisfied
  g(x);       // OK: uses concept map B<float>
  g(y);       // OK: implicitly defines and uses concept map B<int>
  g(p);       // error: no implicit definition of concept map B<int X::*>; requirement not satisfied
  h(q);       // okay: instantiates concept map C<T*> with T=int to satisfy requirement C<T>
  i(p);       // error: i can't get no satisfaction; the
              // concept map template D<T> does not apply
              // because B<int X::*> is not satisfied
}
```

— *end example*]

3 A negative requirement is satisfied if concept map lookup fails to find a concept map that would satisfy the corresponding concept requirement. [*Note:* If concept map lookup results in an ambiguity, concept map lookup halts and the negative requirement is not satisfied. — *end note*] [*Example:*

```
concept A<typename T> { }
auto concept B<typename T> { T operator+(T, T); }

concept_map A<float> { }
concept_map B<float> { }

template<typename T> requires !A<T> void f(T);
template<typename T> requires !B<T> void g(T);

struct X { };
void h(float x, int y, int X::* p) {
  f(x);       // error: concept map A<float> has been defined
  f(y);       // OK: no concept map A<int>
  g(x);       // error: concept map B<float> has been defined
  g(y);       // error: implicitly defines concept map B<int>, requirement not satisfied
  g(p);       // OK: concept map B<int X::*> cannot be implicitly defined
}
```

— *end example*]

4 *Concept map lookup* is a mechanism that attempts to find a concept map that corresponds to the concept instance (call it `I`) formed from the concept of a requirement and its template argument list after substitution of template arguments for their corresponding template parameters. There is an associated full concept name (14.9.1) of `I`; call it `N`. Concept map lookup searches an ordered sequence `Q` (defined below) where each element is a set of concept maps called `S`. For each element in `Q` (progressing from the lowest to the highest-numbered element of `Q`), concept map lookup attempts to find within `S`

— exactly one matching non-template concept map or, if one does not exist,

— exactly one most-specific matching concept map template according to concept map matching rules (14.5.8).

If no matching concept map is found within a set `S` in `Q`, concept map lookup proceeds to the next set in `Q`. If partial ordering of concept map templates results in an ambiguity, concept map lookup returns no result.

5 When concept map lookup is performed during template argument deduction during partial ordering (14.8.2.4), `Q` is defined as containing only one element `S`, where `S` is the set of concept map archetypes synthesized from the requirements of the argument template (14.5.6.2,14.8.2.4).

6 When concept map lookup is performed during the instantiation of a constrained template (14.10.4), `Q` is defined as the following ordered sequence:

1. `S` is the set of concept maps, each with name `N`, that have replaced the concept map archetypes used in the constrained template.
2. `S` is the set of concept maps and concept map templates found by searching for `N` in the namespaces of which a subset of the concept maps in (1) (described below) are members and in the associated namespaces of those namespaces (7.3.1); *using-directives* in those namespaces are not followed during this search. Concept maps in (1) that were implicitly generated from an `auto` concept are not considered when determining which namespaces to search.
3. If a concept map for `I` can be implicitly defined from an `auto` concept (14.9.2), `S` contains the concept map generated from the `auto` concept. Otherwise, `S` is empty.

[*Example:*

```
concept C<typename T> { }
concept D<typename T> { }

namespace N1 {
  concept_map C<int> { }
  concept_map D<int> { }
}
namespace N2 {
  template<C T> void f(T);                 // #1
  template<C T> requires D<T> void f(T);   // #2
  template<C T> void g(T x) {
    f(x);
  }
  using N1::concept_map C<int>;
  void h() {
    g(1); // inside g's call to f, concept map lookup for D<int> finds N1::D<int>; calls #2
  }
}
```

— *end example*]

7 In all other cases (excluding during the instantiation of a constrained template and template argument deduction during partial ordering), `Q` is defined as the ordered sequence:

1. `S` is formed by performing unqualified name lookup (3.4.1) for `N`.
2. `S` is the set of concept maps and concept map templates found by searching for `N` in the namespace of which the concept of `I` is a member and its associated namespaces (7.3.1); *using-directives* in these namespaces are not followed during this search and all names found by way of *using-declarations* are ignored.
3. If a concept map for `I` can be implicitly defined (14.9.2), `S` contains the implicitly-defined concept map for `I`. Otherwise, `S` is empty.

8 [*Note:* When concept map lookup is performed within a constrained context (14.10), concept map archetypes, whose names are placed at the same scope as template parameters, can be found by unqualified lookup. Concept map lookup does not require the definition of a concept map archetype that it finds. — *end note*]

9 If concept map lookup finds a matching concept map in a set `S` within `Q`, concept map lookup succeeds and the remaining elements of `Q` are ignored. [*Note:* The ordering of name-finding methods in `Q` can effect a kind of "concept map hiding" behavior. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
  concept_map C<int> { } // #1

  template<C T> void f(T x);
}
namespace N2 {
  concept_map N1::C<int> { } // #2

  namespace N3 {
    concept_map N1::C<int> { } // #3

    void g() {
      N1::f(1); // uses #3 to satisfy concept requirement N1::C<int>
    }
  }
}
```

— *end example*] — *end note*]

10 [*Note:* Concept maps declared in the namespace of the concept itself will be found last by concept map lookup. [*Example:*

```
namespace N1 {
  concept C<typename T> { }
  concept_map C<int> { }
}
template<N1::C T> void f(T x);
void g() {
  f(1); // Ok, finds N1::concept_map C<int> because it is in the same namespace as concept N1::C.
}
```

— *end example*] — *end note*]

11 If a concept requirement appears (directly or indirectly) multiple times in the requirements of the template, and if the concept maps (14.9.2) used to satisfy the multiple occurrences of the concept requirement are not the same concept map or are different from the concept map that would be determined by concept map lookup (14.10.1.1), then the template arguments do not satisfy the requirements of the template. [*Note:* This does not necessarily imply that the program is ill-formed. — *end note*] [*Example:*

```
concept A<typename T> { }
concept B<typename T> {
  typename X;
  requires A<X>;
}
concept C<typename T> {
  typename X;
  requires A<X>;
}
namespace N1 {
  concept_map A<int> { }  // #1
  concept_map B<int> { }  // uses #1 to satisfy the requirement for A<int>
}
```

```
namespace N2 {
  concept_map A<int> { }  // #2
  concept_map C<int> { }  // uses #2 to satisfy the requirement for A<int>
}
template<typename T> requires B<T> && C<T>
struct S { };
using N1::concept_map B<int>;
using N2::concept_map C<int>;
S<int> s; // ill-formed, two different concept maps for A<int>, #1 and #2
```

— *end example*]

14.10.1.2 Requirement implication [temp.req.impl]

1 The declaration of a constrained template implies additional template requirements that are available within the body of the template. Template requirements are implied from:

— the type of a constrained function template,

— the template arguments of a constrained class template partial specialization,

— the template arguments of a concept map template,

— the template parameters of a constrained template,

— the requirements of a constrained template (including implied requirements),

— the associated requirements and refinements of a concept, and

— the type of an associated function requirement.

2 A requirement is only implied if that requirement is not already present in the template requirements.

3 For every concept requirement in a template's requirements (including implied requirements), requirements for the refinements and associated requirements of the concept named by the concept instance (14.9.3, 14.9.1.3) are implied.

4 The formation of types within certain parts of the declaration of a constrained template (described above) implies the template requirements needed to ensure that the types themselves are well-formed within any instantiation. The following type constructions imply template requirements:

— For every *template-id* `X<A1, A2, ..., AN>`, where `X` is a constrained template, the requirements of `X` (after substitution of the arguments `A1`, `A2`, ..., `AN`) are implied. [*Example:*

```
template<LessThanComparable T> class set { /* ... */ };

template<CopyConstructible T>
void maybe_add_to_set(std::set<T>& s, const T& value);
// use of std::set<T> implicitly adds requirement LessThanComparable<T>
```

— *end example*]

— For every type *cv* `T` that is used as the type of a variable, where `T` aliases an archetype, the concept requirement `VariableType<T>` is implied.

— For every type "pointer to *cv* `T`", where `T` aliases a type archetype, the concept requirement `PointeeType<T>` is implied.

— For every type "reference to *cv* `T`", where `T` aliases a type archetype, the concept requirement `ReferentType<T>` is implied.

— For every type "pointer to member of `X` of type *cv* `T`":

  — If `X` aliases a type archetype, the concept requirement `ClassType<X>` is implied.

  — If `T` aliases a type archetype, the concept requirement `MemberPointeeType<T>` is implied.

— For every type "array of unknown bound of `T`" or "array of `N T`" where `T` aliases a type archetype, the concept requirement `ValueType<T>` is implied.

— For every type *cv* `T` that occurs as the return type of a function, where `T` aliases an archetype, the concept requirement `Returnable<T>` is implied.

— For every type `T` that is used as the type of a non-type template parameter, where `T` aliases an archetype, the concept requirement `NonTypeTemplateParameterType<T>` is implied.

— For every type `T` that is a parameter type or return type of a `constexpr` function, where `T` aliases an archetype, the concept requirement `LiteralType<T>` is implied.

— For every *qualified-id* that names an associated type or class template, a concept requirement for the concept instance containing that associated type or class template is implied. [*Example:*

```
concept Addable<typename T, typename U> {
  CopyConstructible result_type;
  result_type operator+(T, U);
}

template<CopyConstructible T, CopyConstructible U>
Addable<T, U>::result_type // implies Addable<T, U>
add(T t, U u) {
  return t + u;
}
```

— *end example*]

[*Example:*

```
concept C<typename T> { typename assoc; }

template<typename T>
requires C<T>
C<T>::assoc  // implies Returnable<C<T>::assoc>
f(T*, T&); // implies PointeeType<T> and ReferentType<T>
```

— *end example*]

5 In the definition of a constrained class template partial specialization, the requirements of its primary class template (14.5.5), after substitution of the template arguments of the class template partial specialization, are implied. If this substitution results in a requirement that does not depend on any template parameter, then the requirement must be satisfied (14.10.1); otherwise, the program is ill-formed. [*Example:*

```
template<typename T>
requires EqualityComparable<T>
class simple_set { };

template<std::ObjectType T>
class simple_set<T*> // implies EqualityComparable<T*>
```

```
{
};
```

— *end example*]

6 The template requirements for two templates are *identical* if they contain the same concept, negative, and same-type requirements in arbitrary order. Two requirements are the same if they have the same kind, name the same concept, and have the same template argument lists.

14.10.2 Archetypes [temp.archetype]

1 An *archetype* is a non-dependent type, template, or value whose behavior is defined by the template requirements (14.10.1) of its constrained template. Within a constrained context (14.10), a type, value, or template that has an established archetype (described below) behaves as if it were its archetype. [*Note:* this substitution of archetypes (which are not dependent) for their corresponding types, templates, or values (which would be dependent in an unconstrained template) effectively treats all types, templates, and values (and therefore both expressions and names) in a constrained context as "non-dependent". — *end note*]

2 The archetype of a type is a type, the archetype of a template is a class template, and the archetype of a value is a value.

3 A type in a constrained context aliases an archetype if it is:

- a template type parameter (14.1),
- an associated type (14.9.1.2), or
- a class template specialization involving one or more archetypes.

4 A template in a constrained context aliases an archetype if it is:

- a template template parameter (14.1) or
- an associated class template (14.9.1.2).

5 A value in a constrained context aliases an archetype if it is a *constant-expression* (5.19) whose value depends on a template parameter. Two values that alias archetypes are the same if their expressions are equivalent (14.5.6.1). If the translation of the program requires comparison of two values that alias archetypes and are functionally equivalent but not equivalent, the program is ill-formed, no diagnostic required.

6 If two types, `T1` and `T2`, both alias archetypes and are the same (e.g., due to one or more same-type requirements (14.10.1)), then `T1` and `T2` alias the same archetype `T′`. [*Note:* there is no mechanism to specify the relationships between different value archetypes, because such a mechanism would introduce the need for equational reasoning within the translation process. — *end note*]

7 An archetype does not exist until it is *established*. An archetype becomes established under the following circumstances:

- when it is used in the template argument list of a class template specialization or concept instance whose definition is required,
- at the end of the function declarator in a function declaration in which one or more parameters use a type that aliases the archetype, or
- the archetype is used in a context where a complete type is required.

8 If a same-type requirement attempts to make two types equivalent that both alias established archetypes but do not alias the same archetype, the program is ill-formed.

9 In the declaration of a constrained member, member template, or nested class, archetypes are established as if the member were the only member of its class. [*Note:* This means that members that come before the constrained member, member template, or nested class cannot prevent the expression of additional requirements on template parameters from enclosing scopes. — *end note*] [*Example:* Given:

```
concept C1<typename T> { }
concept C2<typename T> { }
concept C3<typename T> { }

template <C1 T> class X {
  requires C2<T> void g(T) {}
  class B {
    void g(T) {}
    requires C3<T> void f(T) {}
  };
};
```

`X::g` is evaluated as if in the context:

```
template <C1 T> class X {
  requires C2<T> void g(T) {}
};
```

and `X::B::g` is evaluated as if in the context:

```
template <C1 T> class X {
  class B {
    requires C3<T> void f(T) {}
  };
};
```

— *end example*]

10 An archetype `T` *has the requirement* of a given concept instance if `T` appears in the template argument list of the concept instance when that concept instance appears in a requirement whose potential scope encloses a region where the archetype `T` is used. [*Example:*

```
concept C<typename T> {}
template <C T> void f( T ) {}
```

Beginning at the declaration of `T` until the closing curly brace of the definition of `f(T)` above, we say that `T` "has the requirement `C<T>`". — *end example*]

11 An archetype is synthesized from the template requirements of the constrained template in which it is defined; see 14.10.2.1.

12 If two associated member function or member function template requirements that name a constructor or destructor for a type `T` have the same signature, the duplicate signature is ignored.

13 If a class template specialization is an archetype that does not appear as a template argument of any explicitly-specified requirement in the template requirements and whose template is not itself an archetype, then the archetype is an instantiated archetype. An *instantiated archetype* is an archetype whose definition is provided by the instantiation of its template with its template arguments (which involve archetypes). The template shall not be an unconstrained template. [*Note:* Partial ordering of class template partial specializations (14.5.5.2) depends on the properties of the archetypes, as defined by the requirements of the constrained template. When the constrained template is instantiated (14.10.4), partial ordering of class

template partial specializations occurs a second time based on the actual template arguments. — *end note*] [*Example:*

```
template<EqualityComparable T>
struct simple_multiset {
  bool includes(const T&);
  void insert(const T&);
  // ...
};

template<LessThanComparable T>
struct simple_multiset<T> { // A
  bool includes(const T&);
  void insert(const T&);
  // ...
};

template<LessThanComparable T>
bool first_access(const T& x) {
  static simple_multiset<T> set; // instantiates simple_multiset<T'>, where T' is the archetype of T ,
                                 // from the partial specialization of simple_multiset marked 'A'
  return set.includes(x)? false : (set.insert(x), true);
}
```

— *end example*]

[*Note:* Class template specializations for which template requirements are specified behave as normal archetypes. — *end note*] [*Example:*

```
auto concept CopyConstructible<typename T> {
  T::T(const T&);
}

template<CopyConstructible T> struct vector;

auto concept VectorLike<typename X> {
  typename value_type = typename X::value_type;
  X::X();
  void X::push_back(const value_type&);
  value_type& X::front();
}

template<CopyConstructible T>
requires VectorLike<vector<T>> // vector<T> is an archetype (but not an instantiated archetype)
void f(const T& value) {
  vector<T> x;        // OK: default constructor in VectorLike<vector<T> >
  x.push_back(value); // OK: push_back in VectorLike<vector<T> >
  VectorLike<vector<T>>::value_type& val = x.front(); // OK: front in VectorLike<vector<T> >
}
```

— *end example*]

14 [*Note:* Constrained class templates involving recursive definitions are ill-formed if the recursive class template specialization is an instantiated archetype. Constrained class templates involving recursive definitions can be specified by adding template requirements on the recursive class template specializations, making them archetypes that are not instantiated archetypes. [*Example:*

```
template<CopyConstructible... T> class tuple;

template<CopyConstructible Head, CopyConstructible... Tail>
class tuple<Head, Tail...> : tuple<Tail...> // ill-formed: tuple<Tail...> is an instantiated archetype,
                                            // but it is an incomplete type
{
  Head head;
  // ...
};

template<> class tuple<> { /* ... */ };
```

— *end example*] — *end note*]

15 In a constrained context, for each concept requirement that is stated in or implied by the corresponding requirements, a *concept map archetype* for that requirement is synthesized by substituting the archetype of `T` for each occurrence of `T` within the template arguments of the requirement. The concept map archetype acts as a concept map, and its definition is used to resolve name lookup into requirements scope (3.3.8) and satisfy the requirements of templates used inside the constrained context. When the definition of a concept map archetype is required, it is synthesized from the definition of its corresponding concept (14.9.2). An implementation shall not define a concept map archetype unless the definition of that concept map archetype is required. [*Example:*

```
concept SignedIntegral<typename T> {
  T::T(const T&);
  T operator-(T);
}
concept RandomAccessIterator<typename T> {
  SignedIntegral difference_type;
  difference_type operator-(T, T);
}
template<SignedIntegral T> T negate(const T& t) { return -t; }

template<RandomAccessIterator Iter>
RandomAccessIterator<Iter>::difference_type distance(Iter f, Iter l) {
  typedef RandomAccessIterator<Iter>::difference_type D;
  D dist = f - l;      // OK: - operator resolves to synthesized operator- in
                       // the concept map archetype RandomAccessIterator<Iter'>,
                       // where Iter' is the archetype of Iter
  return negate(dist);// OK, concept map archetype RandomAccessIterator<Iter'>
                       // implies the concept map archetype SignedIntegral<D'>,
                       // where D' is the archetype of D
```

— *end example*]

14.10.2.1 Assembling archetypes [temp.archetype.assemble]

1 A type archetype is a unique generated class type whose special member functions are each either user-provided or deleted (depending on template requirements) and whose other members are given entirely by its template requirements. A type archetype is considered to be completely defined when it is established (although it may later acquire additional members as a result of additional requirements).

2 The archetype `T'` of `T` contains a public member function or member function template corresponding to each member function or member function template of each concept map archetype corresponding to a concept requirement that names `T` (14.10.1). [*Example:*

```
concept CopyConstructible<typename T> {
  T::T(const T&);
}

concept MemSwappable<typename T> {
  void T::swap(T&);
}

template<typename T>
requires CopyConstructible<T> && MemSwappable<T>
void foo(T& x) {
  // archetype T′ of T contains a copy constructor T′::T′(const T′&) from CopyConstructible<T>
  // and a member function void swap(T′&) from MemSwappable<T>
  T y(x);
  y.swap(x);
}
```

— *end example*]

3 If no requirement specifies a copy constructor for a type `T`, a copy constructor is implicitly declared (12.8) in the archetype of `T` with the following signature:

```
T(const T&) = delete;
```

[*Example:*

```
concept DefaultConstructible<typename T> {
  T::T();
}

concept MoveConstructible<typename T> {
  T::T(T&&);
}

template<typename T>
requires DefaultConstructible<T> && MoveConstructible<T>
void f(T x) {
  T y = T();  // OK: move-constructs y from default-constructed T
  T z(x);     // error: overload resolution selects implicitly-declared
              // copy constructor, which is deleted
}
```

— *end example*]

4 If no requirement specifies a copy assignment operator for a type `T`, a copy assignment operator is implicitly declared (12.8) in the archetype of `T` with the following signature:

```
T& T::operator=(const T&) = delete;
```

5 If no requirement specifies a destructor for a type `T`, a destructor is implicitly declared (12.4) in the archetype of `T` with the following signature:

```
~T() = delete;
```

6 If no requirement specifies a unary `&` operator for a type `T`, a unary member operator `&` is implicitly declared in the archetype of `T` for each *cv* that is a valid *cv-qualifier-seq*:

```
cv T* operator&() cv = delete;
```

7 For each of the allocation functions `new`, `new[]`, `delete`, and `delete[]` (12.5), if no requirement specifies the corresponding operator with a signature below, that allocation function is implicitly declared as a member function in the archetype `T′` of `T` with the corresponding signature from the following list:

```
static void* T′::operator new(std::size_t) = delete;
static void* T′::operator new(std::size_t, void*) = delete;
static void* T′::operator new(std::size_t, const std::nothrow_t&) throw() = delete;
static void* T′::operator new[](std::size_t) = delete;
static void* T′::operator new[](std::size_t, void*) = delete;
static void* T′::operator new[](std::size_t, const std::nothrow_t&) throw() = delete;
static void T′::operator delete(void*) = delete;
static void T′::operator delete(void*, void*) = delete;
static void T′::operator delete(void*, const std::nothrow_t&) throw() = delete;
static void T′::operator delete[](void*) = delete;
static void T′::operator delete[](void*, void*) = delete;
static void T′::operator delete[](void*, const std::nothrow_t&) throw() = delete;
```

8 If the template requirements contain a requirement `std::DerivedFrom<T, Base>`, then the archetype of `T` is publicly derived from the archetype of `Base`. If the same `std::DerivedFrom<T, Base>` requirement occurs more than once within the template requirements, the repeated `std::DerivedFrom<T, Base>` requirements are ignored.

14.10.3 Candidate sets [temp.constrained.set]

1 A *candidate set* is a set containing functions and function templates that is determined by a *seed* (defined below). A candidate set is defined in a constrained context (a *retained candidate set*, 14.10.4) or as the result of satisfying an associated function requirement in a concept map (an *associated function candidate set*, 14.9.2.1). Candidate sets are used to capture a set of candidate functions that are used in the instantiation of a constrained template (14.10.4) or when referring to members in a concept map (14.9.2). [*Note:* For the purposes of this section, candidate operator functions (13.6) are considered functions. — *end note*]

2 Each candidate set has a *seed*, which provides the basis for the candidate set itself. All functions and function templates that are consistent with the seed are contained in the candidate set. The seed is determined as part of the definition of the candidate set, and will be one of:

- a function,
- the initialization of an object (8.5), or
- a *pseudo-destructor-name* (5.2.4).

3 A function is *consistent with* the seed if all of the following apply:

- it has the same name as the seed,
- its enclosing namespace is the same as the enclosing namespace of the seed,
- the seed has a return type of *cv* `void` or the function has the same return type as the seed, after the reference (if any) and then top-level *cv-qualifiers* (if any) have been removed from the return types of the seed and the function, and
- it has the same *parameter-type-list* as the seed, after making the following adjustments to both *parameter-type-list* s:
  - for a non-static member function, add the implicit object parameter (13.3.1) as the first parameter in the *parameter-type-list*,

— for each parameter type, remove the top-level reference (if any) and then top-level *cv-qualifiers* (if any),

— if the function has M parameters, the seed has N parameters, and $M > N$, remove each of the last $M - N$ parameters that has a default argument from the *parameter-type-list*, and

— remove the ellipsis, if any.

[*Note:* No function or function template is consistent with a non-function seed. A seed that is a function is consistent with itself. — *end note*]

4 A function template is consistent with the seed if all of the following apply:

— it has the same name as the seed,

— its enclosing namespace is the same as the enclosing namespace of the seed, and

— if the candidate set containing the seed is a retained candidate set, the function template is a constrained function template.

5 A candidate set is a set of overloaded functions. Overload resolution (13.3) for a candidate set is subject to the following additional conditions:

— the set of candidate functions for overload resolution is the set of functions in the candidate set, and

— if template argument deduction on a candidate function produces a function template specialization that is not consistent with the seed of the candidate set, the function template specialization is not a viable function (13.3.2).

14.10.4 Instantiation of constrained templates [temp.constrained.inst]

1 Instantiation of a constrained template replaces each template parameter within the definition of the template with its corresponding template argument, using the same process as for unconstrained templates (14.7).

2 Instantiation of a constrained template also replaces each concept map archetype with the concept map that satisfied the corresponding template requirement. [*Note:* A concept member that had resolved to a member of a concept map archetype now refers to a member of the corresponding concept map. — *end note*]

3 In the instantiation of a constrained template, a call to a function that resolves to an associated function in a concept map archetype (14.10.2) is instantiated using the associated function candidate set (14.10.3) that satisfies the corresponding associated function requirement in the concept map that replaces the concept map archetype. The instantiated form depends on the seed of the associated function candidate set and the syntactic form that resulted in the call in the constrained template:

— If the seed is a function, the instantiated form is a call to the associated function candidate set. [*Example:*

```
concept F<typename T> {
  T::T();
  void f(T const&);
}

template<typename T> requires F<T>
void g(T const& x) {
  f(x);       // calls F<T>::f. When instantiated with T=X, calls #1
  f(T());     // calls F<T>::f. When instantiated with T=X, calls #2
}
```

```
struct X {};
void f(X const&);       // #1
void f(X&&);            // #2

concept_map F<X> { }    // associated function candidate set for
                        // f(X const&) contains #1 and #2, seed is #1

void h(X const& x) {
  g(x);
}
```

— *end example*]

— Otherwise, if the seed is a *pseudo-destructor-name*, the instantiated form is a pseudo destructor call (5.2.4).

— Otherwise, if the seed is the initialization of an object, the instantiated form is the appropriate kind of initialization (zero-initialization, default-initialization, value-initialization, or no initialization) based on the syntactic form in the constrained template. [*Example:*

```
auto concept DefaultConstructible<typename T> {
  T::T();
}

template<DefaultConstructible T>
void f() {
  T t1;   // type-checks as a call to DefaultConstructible<T>'s T::T()
  T t2{}; // type-checks as a call to DefaultConstructible<T>'s T::T()
}

template void f<int>(); // t1 is not initialized, t2 is value-initialized
```

— *end example*]

4 A use of a function template specialization in a constrained template instantiates to a reference to that function template specialization's retained candidate set. The *retained candidate set* is a candidate set (14.10.3) whose seed is the function template specialization. [*Example:*

```
concept InputIterator<typename Iter> {
  typename difference_type;
}
concept BidirectionalIterator<typename Iter> : InputIterator<Iter> { }
concept RandomAccessIterator<typename Iter> : BidirectionalIterator<Iter> { }

template<InputIterator Iter>
  void advance(Iter& i, Iter::difference_type n);     // #1
template<BidirectionalIterator Iter>
  void advance(Iter& i, Iter::difference_type n);     // #2

template<BidirectionalIterator Iter> void f(Iter i) {
  advance(i, 1);        // seed function is #2
}

concept_map RandomAccessIterator<int*> {
  typedef std::ptrdiff_t difference_type;
}
```

```
template<RandomAccessIterator Iter>
  void advance(Iter& i, Iter::difference_type n);      // #3

void g(int* i) {
  f(i);                   // in the call to advance(), #2 is the seed of the
                          // retained candidate set, the retained candidate set
                          // contains #1, #2, and #3, and partial ordering of
                          // function templates selects #3.
}
```

— *end example*] — *end note*]

5 When a function call in the instantiation of a constrained template is instantiated as a call to a candidate set (14.10.3), the user-defined conversions applied to the function call arguments in the call to the seed are applied in the call to the candidate set in the instantiation. When such a user-defined conversion is applied to an instantiated function call argument, it acts as the user-defined conversion sequence (13.3.3.1.2) for that argument. [*Note:* the user-defined conversion sequence may still be preceded and followed by a standard conversion sequence. — *end note*] [*Example:*

```
concept C<typename T> {
  operator int(const T&);
};

class A { };

concept_map C<A> {
  operator int(const A&) { return 0; }
}

void f(int);

template<C T> void f(const T& x) {
  f(x);       // OK: implicit, user-defined conversion from T to int
}

template void f<A>(const A&); // OK: call f(x) applies the implicit, user-defined
                              // conversion from the template definition using C<A>::operator int.
```

— *end example*]

6 In the instantiation of a constrained template, a template specialization whose template arguments involve the constrained template's template parameters (14.10.2) is replaced by the template specialization that results from substituting the constrained template's template arguments for their corresponding template parameters. [*Note:* If the template specialization is a template alias (14.5.7), the substitution occurs in the *type-id* of the template alias. — *end note*] The resulting type (call it A) shall be compatible with the type involving the template parameters (call it A′) that it replaced, otherwise the program is ill-formed. The template specializations are compatible if all of the following conditions hold:

— for each function, function template, or data member m of A′ referenced by the constrained template, there exists a member named m in A that is accessible from the constrained template and whose type, storage specifiers, template parameters (if any), and template requirements (if any) are the same as the those of A′::m after substituting the constrained template's template arguments for the corresponding template parameters,

— for each member type `t` of `A′` referenced by the constrained template, there exists a member type `t` in `A` that is accessible from the constrained template and is compatible with the member type `A′::t` as specified herein, and

— for each base class `B′` of `A′` referenced by a derived-to-base conversion (4.10) in the constrained template, there exists an unambiguous base class `B` of `A` that is accessible from the constrained template, where `B` is the type produced by substituting the constrained template's template parameters with the corresponding template arguments in `B′`.

[*Example:*

```
auto concept CopyConstructible<typename T> {
  T::T(const T&);
}

template<CopyConstructible T>
struct vector { // A
  vector(int, T const &);
  T& front();
};

template<typename T>
struct vector<T*> { // B
  vector(int, T* const &);
  T*& front();
};

template<>
struct vector<bool> { // C
  vector(int, bool);
  bool front();
};

template<CopyConstructible T>
void f(const T& x) {
  vector<T> vec(1, x);
  T& ref = vec.front();
}

void g(int i, int* ip, bool b) {
  f(i);       // OK: instantiation of f<int> uses vector<int>, instantiated from A
  f(ip);      // OK: instantiation of f<int*> uses vector<int*>, instantiated from B
  f(b);       // ill-formed, detected in the instantiation of f<bool>, which uses the vector<bool> specialization
C:
              // vector<bool>::front is not compatible with vector<T>::front (where T is bool)
}
```

— *end example*]

15 Exception handling [except]

1 Exception handling provides a way of transferring control and information from a point in the execution of a program to an exception handler associated with a point previously passed by the execution. A handler will be invoked only by a *throw-expression* invoked in code executed in the handler's try block or in functions called from the handler's try block .

try-block:
`try` *compound-statement handler-seq*

function-try-block:
`try` *ctor-initializer*$_{opt}$ *compound-statement handler-seq*

handler-seq:
handler handler-seq$_{opt}$

handler:
`catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
type-specifier-seq declarator
type-specifier-seq abstract-declarator
type-specifier-seq
`...`

throw-expression:
`throw` *assignment-expression*$_{opt}$

2 A *try-block* is a *statement* (Clause 6). A *throw-expression* is of type `void`. Code that executes a *throw-expression* is said to "throw an exception;" code that subsequently gets control is called a "handler." [*Note:* within this Clause "try block" is taken to mean both *try-block* and *function-try-block*. — *end note*]

3 A `goto` or `switch` statement shall not be used to transfer control into a try block or into a handler. [*Example:*

```
void f() {
  goto l1;          // Ill-formed
  goto l2;          // Ill-formed
  try {
    goto l1;        // OK
    goto l2;        // Ill-formed
    l1: ;
  } catch (...) {
    l2: ;
    goto l1;        // Ill-formed
    goto l2;        // OK
  }
}
```

— *end example*] A `goto`, `break`, `return`, or `continue` statement can be used to transfer control out of a try block or handler. When this happens, each variable declared in the try block will be destroyed in the context that directly contains its declaration. [*Example:*

```
lab:  try {
  T1 t1;
```

```
try {
  T2 t2;
  if (condition)
    goto lab;
  } catch(...) { /* handler 2 */ }
} catch(...) { /* handler 1 */ }
```

Here, executing `goto lab;` will destroy first `t2`, then `t1`, assuming the *condition* does not declare a variable. Any exception raised while destroying `t2` will result in executing *handler 2*; any exception raised while destroying `t1` will result in executing *handler 1*. — *end example*]

4 A *function-try-block* associates a *handler-seq* with the *ctor-initializer*, if present, and the *compound-statement*. An exception thrown during the execution of the initializer expressions in the *ctor-initializer* or during the execution of the *compound-statement* transfers control to a handler in a *function-try-block* in the same way as an exception thrown during the execution of a *try-block* transfers control to other handlers. [*Example:*

```
int f(int);
class C {
  int i;
  double d;
public:
  C(int, double);
};

C::C(int ii, double id)
try : i(f(ii)), d(id) {
    // constructor statements
}
catch (...) {
    // handles exceptions thrown from the ctor-initializer
    // and from the constructor statements
}
```

— *end example*]

15.1 Throwing an exception [except.throw]

1 Throwing an exception transfers control to a handler. An object is passed and the type of that object determines which handlers can catch it. [*Example:*

```
throw "Help!";
```

can be caught by a *handler* of `const char*` type:

```
try {
    // ...
}
catch(const char* p) {
    // handle character string exceptions here
}
```

and

```
class Overflow {
public:
    Overflow(char,double,double);
};
```

```
void f(double x) {
    throw Overflow('+',x,3.45e107);
}
```

can be caught by a handler for exceptions of type `Overflow`

```
try {
    f(1.2);
} catch(Overflow& oo) {
    // handle exceptions of type Overflow here
}
```

— *end example*]

2 When an exception is thrown, control is transferred to the nearest handler with a matching type (15.3); "nearest" means the handler for which the *compound-statement* or *ctor-initializer* following the `try` keyword was most recently entered by the thread of control and not yet exited.

3 A *throw-expression* initializes a temporary object, called the *exception object*, the type of which is determined by removing any top-level *cv-qualifier*s from the static type of the operand of `throw` and adjusting the type from "array of `T`" or "function returning `T`" to "pointer to `T`" or "pointer to function returning `T`", respectively. [*Note:* the temporary object created for a *throw-expression* that is a string literal is never of type `char*`, `char16_t*`, `char32_t*`, or `wchar_t*`; that is, the special conversions for string literals from the types "array of `const char`", "array of `const char16_t`", "array of `const char32_t`", and "array of `const wchar_t`" to the types "pointer to `char`", "pointer to `char16_t`", "pointer to `char32_t`", and "pointer to `wchar_t`", respectively (4.2), are never applied to a *throw-expression*. — *end note*] The temporary is an lvalue and is used to initialize the variable named in the matching *handler* (15.3). The type of the *throw-expression* shall not be an incomplete type, or a pointer to an incomplete type other than (possibly cv-qualified) `void`. Except for these restrictions and the restrictions on type matching mentioned in 15.3, the operand of `throw` is treated exactly as a function argument in a call (5.2.2) or the operand of a return statement.

4 The memory for the temporary copy of the exception being thrown is allocated in an unspecified way, except as noted in 3.7.4.1. The temporary persists as long as there is a handler being executed for that exception. In particular, if a handler exits by executing a `throw;` statement, that passes control to another handler for the same exception, so the temporary remains. When the last remaining active handler for the exception exits by any means other than `throw;` the temporary object is destroyed and the implementation may deallocate the memory for the temporary object; any such deallocation is done in an unspecified way. The destruction occurs immediately after the destruction of the object declared in the *exception-declaration* in the handler.

5 When the thrown object is a class object, the copy constructor and the destructor shall be accessible, even if the copy operation is elided (12.8).

6 An exception is considered caught when a handler for that exception becomes active (15.3). [*Note:* an exception can have active handlers and still be considered uncaught if it is rethrown. — *end note*]

7 A *throw-expression* with no operand rethrows the currently handled exception (15.3). The exception is reactivated with the existing temporary; no new temporary exception object is created. The exception is no longer considered to be caught; therefore, the value of `std::uncaught_exception()` will again be `true`. [*Example:* code that must be executed because of an exception yet cannot completely handle the exception can be written like this:

```
try {
    // ...
} catch (...) {      // catch all exceptions
  // respond (partially) to exception
```

```
    throw;              // pass the exception to some
                        // other handler
}
```

— *end example*]

8 If no exception is presently being handled, executing a *throw-expression* with no operand calls `std::terminate()` (15.5.1).

15.2 Constructors and destructors [except.ctor]

1 As control passes from a *throw-expression* to a handler, destructors are invoked for all automatic objects constructed since the try block was entered. The automatic objects are destroyed in the reverse order of the completion of their construction.

2 An object that is partially constructed or partially destroyed will have destructors executed for all of its fully constructed base classes and non-variant members, that is, for subobjects for which the principal constructor (12.6.2) has completed execution and the destructor has not yet begun execution. Similarly, if the non-delegating constructor for an object has completed execution and a delegating constructor for that object exits with an exception, the object's destructor will be invoked. If the object was allocated in a *new-expression*, the matching deallocation function (3.7.4.2, 5.3.4, 12.5), if any, is called to free the storage occupied by the object.

3 The process of calling destructors for automatic objects constructed on the path from a try block to a *throw-expression* is called "*stack unwinding*." [*Note:* If a destructor called during stack unwinding exits with an exception, `std::terminate` is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor. — *end note*]

15.3 Handling an exception [except.handle]

1 The *exception-declaration* in a *handler* describes the type(s) of exceptions that can cause that *handler* to be entered. The *exception-declaration* shall not denote an incomplete type. The *exception-declaration* shall not denote a pointer or reference to an incomplete type, other than `void*`, `const void*`, `volatile void*`, or `const volatile void*`.

2 A handler of type "array of `T`" or "function returning `T`" is adjusted to be of type "pointer to `T`" or "pointer to function returning `T`", respectively.

3 A *handler* is a match for an exception object of type `E` if

- The *handler* is of type *cv* `T` or *cv* `T&` and `E` and `T` are the same type (ignoring the top-level *cv-qualifiers*), or
- the *handler* is of type *cv* `T` or *cv* `T&` and `T` is an unambiguous public base class of `E`, or
- the *handler* is of type *cv1* `T*` *cv2* and `E` is a pointer type that can be converted to the type of the *handler* by either or both of
  - a standard pointer conversion (4.10) not involving conversions to pointers to private or protected or ambiguous classes
  - a qualification conversion
- the *handler* is a pointer or pointer to member type and `E` is `std::nullptr_t`.

[*Note:* a *throw-expression* whose operand is an integral constant expression of integer type that evaluates to zero does not match a handler of pointer or pointer to member type. — *end note*]

[*Example:*

```
class Matherr { /* ... */ virtual void vf(); };
class Overflow: public Matherr { /* ... */ };
class Underflow: public Matherr { /* ... */ };
class Zerodivide: public Matherr { /* ... */ };

void f() {
  try {
    g();
  } catch (Overflow oo) {
        // ...
  } catch (Matherr mm) {
        // ...
  }
}
```

Here, the `Overflow` handler will catch exceptions of type `Overflow` and the `Matherr` handler will catch exceptions of type `Matherr` and of all types publicly derived from `Matherr` including exceptions of type `Underflow` and `Zerodivide`. — *end example*]

4 The handlers for a try block are tried in order of appearance. That makes it possible to write handlers that can never be executed, for example by placing a handler for a derived class after a handler for a corresponding base class.

5 A `...` in a handler's *exception-declaration* functions similarly to `...` in a function parameter declaration; it specifies a match for any exception. If present, a `...` handler shall be the last handler for its try block.

6 If no match is found among the handlers for a try block, the search for a matching handler continues in a dynamically surrounding try block.

7 A handler is considered active when initialization is complete for the formal parameter (if any) of the catch Clause. [*Note:* the stack will have been unwound at that point. — *end note*] Also, an implicit handler is considered active when `std::terminate()` or `std::unexpected()` is entered due to a throw. A handler is no longer considered active when the catch Clause exits or when `std::unexpected()` exits after being entered due to a throw.

8 The exception with the most recently activated handler that is still active is called the *currently handled exception*.

9 If no matching handler is found, the function `std::terminate()` is called; whether or not the stack is unwound before this call to `std::terminate()` is implementation-defined (15.5.1).

10 Referring to any non-static member or base class of an object in the handler for a *function-try-block* of a constructor or destructor for that object results in undefined behavior.

11 The fully constructed base classes and members of an object shall be destroyed before entering the handler of a *function-try-block* of a constructor for that object. Similarly, if a delegating constructor for an object exits with an exception after the non-delegating constructor for that object has completed execution, the object's destructor shall be executed before entering the handler of a *function-try-block* of a constructor for that object. The base classes and non-variant members of an object shall be destroyed before entering the handler of a *function-try-block* of a destructor for that object (12.4).

12 The scope and lifetime of the parameters of a function or constructor extend into the handlers of a *function-try-block*.

13 Exceptions thrown in destructors of objects with static storage duration or in constructors of namespace-scope objects with static storage duration are not caught by a *function-try-block* on `main()`. Exceptions

thrown in destructors of objects with thread storage duration or in constructors of namespace-scope objects with thread storage duration are not caught by a *function-try-block* on the initial function of the thread.

14 If a return statement appears in a handler of the *function-try-block* of a constructor, the program is ill-formed.

15 The currently handled exception is rethrown if control reaches the end of a handler of the *function-try-block* of a constructor or destructor. Otherwise, a function returns when control reaches the end of a handler for the *function-try-block* (6.6.3). Flowing off the end of a *function-try-block* is equivalent to a `return` with no value; this results in undefined behavior in a value-returning function (6.6.3).

16 When the *exception-declaration* specifies a class type, a copy constructor is used to initialize either the object declared in the *exception-declaration* or, if the *exception-declaration* does not specify a name, a temporary object of that type. The object shall not have an abstract class type. The object is destroyed when the handler exits, after the destruction of any automatic objects initialized within the handler. The copy constructor and destructor shall be accessible in the context of the handler. If the copy constructor and destructor are implicitly declared (12.8), such a use in the handler causes these functions to be implicitly defined; otherwise, the program shall provide a definition for these functions.

17 The copy constructor and destructor associated with the object shall be accessible even if the copy operation is elided (12.8).

18 When the handler declares a non-constant object, any changes to that object will not affect the temporary object that was initialized by execution of the *throw-expression*. When the handler declares a reference to a non-constant object, any changes to the referenced object are changes to the temporary object initialized when the *throw-expression* was executed and will have effect should that object be rethrown.

15.4 Exception specifications [except.spec]

1 A function declaration lists exceptions that its function might directly or indirectly throw by using an *exception-specification* as a suffix of its declarator.

exception-specification:
`throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
type-id `...`$_{opt}$
type-id-list `,` *type-id* `...`$_{opt}$

2 An *exception-specification* shall appear only on a function declarator for a function type, pointer to function type, reference to function type, or pointer to member function type that is the top-level type of a declaration or definition, or on such a type appearing as a parameter or return type in a function declarator. An *exception-specification* shall not appear in a typedef declaration. [*Example:*

```
void f() throw(int);                    // OK
void (*fp)() throw (int);               // OK
void g(void pfa() throw(int));          // OK
typedef int (*pf)() throw(int);         // ill-formed
```

— *end example*] A type denoted in an *exception-specification* shall not denote an incomplete type. A type denoted in an *exception-specification* shall not denote a pointer or reference to an incomplete type, other than `void*`, `const void*`, `volatile void*`, or `const volatile void*`.

3 If any declaration of a function has an *exception-specification*, all declarations, including the definition and an explicit specialization, of that function shall have an *exception-specification* with the same set of *type-id*s. If any declaration of a pointer to function, reference to function, or pointer to member function has an *exception-specification*, all occurrences of that declaration shall have an *exception-specification* with the same set of *type-id*s. In an explicit instantiation an *exception-specification* may be specified, but is not

required. If an *exception-specification* is specified in an explicit instantiation directive, it shall have the same set of *type-id*s as other declarations of that function. A diagnostic is required only if the sets of *type-id*s are different within a single translation unit.

4 If a virtual function has an *exception-specification*, all declarations, including the definition, of any function that overrides that virtual function in any derived class shall only allow exceptions that are allowed by the *exception-specification* of the base class virtual function. [*Example:*

```
struct B {
  virtual void f() throw (int, double);
  virtual void g();
};

struct D: B {
  void f();                       // ill-formed
  void g() throw (int);           // OK
};
```

The declaration of `D::f` is ill-formed because it allows all exceptions, whereas `B::f` allows only `int` and `double`. — *end example*] A similar restriction applies to assignment to and initialization of pointers to functions, pointers to member functions, and references to functions: the target entity shall allow at least the exceptions allowed by the source value in the assignment or initialization. [*Example:*

```
class A { /* ... */ };
void (*pf1)();      // no exception specification
void (*pf2)() throw(A);

void f() {
  pf1 = pf2;        // OK: pf1 is less restrictive
  pf2 = pf1;        // error: pf2 is more restrictive
}
```

— *end example*]

5 In such an assignment or initialization, *exception-specification*s on return types and parameter types shall match exactly. In other assignments or initializations, *exception-specification*s shall match exactly.

6 An *exception-specification* can include the same type more than once and can include classes that are related by inheritance, even though doing so is redundant. An *exception-specification* can also include the class `std::bad_exception` (18.7.2.1).

7 A function is said to *allow* an exception of type `E` if its *exception-specification* contains a type `T` for which a handler of type `T` would be a match (15.3) for an exception of type `E`.

8 Whenever an exception is thrown and the search for a handler (15.3) encounters the outermost block of a function with an *exception-specification*, the function `std::unexpected()` is called (15.5.2) if the *exception-specification* does not allow the exception. [*Example:*

```
class X { };
class Y { };
class Z: public X { };
class W { };

void f() throw (X, Y) {
  int n = 0;
  if (n) throw X();             // OK
  if (n) throw Z();             // also OK
```

```
  throw W();                    // will call std::unexpected()
}
```

— *end example*]

9 The function `std::unexpected()` may throw an exception that will satisfy the *exception-specification* for which it was invoked, and in this case the search for another handler will continue at the call of the function with this *exception-specification* (see 15.5.2), or it may call `std::terminate()`.

10 An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow. [*Example:*

```
extern void f() throw(X, Y);

void g() throw(X) {
  f();                          // OK
}
```

the call to `f` is well-formed even though when called, `f` might throw exception `Y` that `g` does not allow. — *end example*]

11 A function with no *exception-specification* allows all exceptions. A function with an empty *exception-specification*, `throw()`, does not allow any exceptions.

12 An *exception-specification* is not considered part of a function's type.

13 An implicitly declared special member function (Clause 12) shall have an *exception-specification*. If `f` is an implicitly declared default constructor, copy constructor, destructor, or copy assignment operator, its implicit *exception-specification* specifies the *type-id* `T` if and only if `T` is allowed by the *exception-specification* of a function directly invoked by `f`'s implicit definition; `f` shall allow all exceptions if any function it directly invokes allows all exceptions, and `f` shall allow no exceptions if every function it directly invokes allows no exceptions. [*Example:*

```
struct A {
  A();
  A(const A&) throw();
  ~A() throw(X);
};
struct B {
  B() throw();
  B(const B&) throw();
  ~B() throw(Y);
};
struct D : public A, public B {
    // Implicit declaration of D::D();
    // Implicit declaration of D::D(const D&) throw();
    // Implicit declaration of D::~D() throw(X,Y);
};
```

Furthermore, if `A::~A()` or `B::~B()` were virtual, `D::~D()` would not be as restrictive as that of `A::~A`, and the program would be ill-formed since a function that overrides a virtual function from a base class shall have an *exception-specification* at least as restrictive as that in the base class. — *end example*]

14 In an *exception-specification*, a *type-id* followed by an ellipsis is a pack expansion (14.5.3).

15.5 Special functions [except.special]

1 The exception handling mechanism relies on two functions, `std::terminate()` and `std::unexpected()`, for coping with errors related to the exception handling mechanism itself (18.7).

15.5.1 The `std::terminate()` function [except.terminate]

1 In the following situations exception handling must be abandoned for less subtle error handling techniques:

- when the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught (15.1), calls a user function that exits via an uncaught exception,[138]
- when the exception handling mechanism cannot find a handler for a thrown exception (15.3), or
- when the destruction of an object during stack unwinding (15.2) exits using an exception, or
- when construction or destruction of a non-local object with static or thread storage duration exits using an exception (3.6.2), or
- when execution of a function registered with `std::atexit` exits using an exception (18.4), or
- when a *throw-expression* with no operand attempts to rethrow an exception and no exception is being handled (15.1), or
- when `std::unexpected` throws an exception which is not allowed by the previously violated *exception-specification*, and `std::bad_exception` is not included in that *exception-specification* (15.5.2), or
- when the implementation's default unexpected exception handler is called (18.7.2.2).

2 In such cases, `std::terminate()` is called (18.7.3). In the situation where no matching handler is found, it is implementation-defined whether or not the stack is unwound before `std::terminate()` is called. In all other situations, the stack shall not be unwound before `std::terminate()` is called. An implementation is not permitted to finish stack unwinding prematurely based on a determination that the unwind process will eventually cause a call to `std::terminate()`.

15.5.2 The `std::unexpected()` function [except.unexpected]

1 If a function with an *exception-specification* throws an exception that is not listed in the *exception-specification*, the function `std::unexpected()` is called (18.7.2) immediately after completing the stack unwinding for the former function.

2 The `std::unexpected()` function shall not return, but it can throw (or re-throw) an exception. If it throws a new exception which is allowed by the exception specification which previously was violated, then the search for another handler will continue at the call of the function whose exception specification was violated. If it throws or rethrows an exception that the *exception-specification* does not allow then the following happens: If the *exception-specification* does not include the class `std::bad_exception` (18.7.2.1) then the function `std::terminate()` is called, otherwise the thrown exception is replaced by an implementation-defined object of the type `std::bad_exception` and the search for another handler will continue at the call of the function whose *exception-specification* was violated.

138) For example, if the object being thrown is of a class with a copy constructor, `std::terminate()` will be called if that copy constructor exits with an exception during a `throw`.

3 Thus, an *exception-specification* guarantees that only the listed exceptions will be thrown. If the *exception-specification* includes the type `std::bad_exception` then any exception not on the list may be replaced by `std::bad_exception` within the function `std::unexpected()`.

15.5.3 The `std::uncaught_exception()` function [except.uncaught]

1 The function `std::uncaught_exception()` returns `true` after completing evaluation of the object to be thrown until completing the initialization of the *exception-declaration* in the matching handler (18.7.4). This includes stack unwinding. If the exception is rethrown (15.1), `std::uncaught_exception()` returns `true` from the point of rethrow until the rethrown exception is caught again.

15.6 Exceptions and access [except.access]

1 If the *exception-declaration* in a catch Clause has class type, and the function in which the catch Clause occurs does not have access to the destructor of that class, the program is ill-formed.

2 An object can be thrown if it can be copied and destroyed in the context of the function in which the *throw-expression* occurs.

16 Preprocessing directives [cpp]

1 A preprocessing directive consists of a sequence of preprocessing tokens. The first token in the sequence is a # preprocessing token that (at the start of translation phase 4) is either the first character in the source file (optionally after white space containing no new-line characters) or that follows white space containing at least one new-line character. The last token in the sequence is the first new-line character that follows the first token in the sequence.[139] A new-line character ends the preprocessing directive even if it occurs within what would otherwise be an invocation of a function-like macro.

preprocessing-file:
group$_{opt}$

group:
group-part
group group-part

group-part:
if-section
control-line
text-line
 # non-directive

if-section:
if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
`# if` *constant-expression new-line group*$_{opt}$
`# ifdef` *identifier new-line group*$_{opt}$
`# ifndef` *identifier new-line group*$_{opt}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
`# elif` *constant-expression new-line group*$_{opt}$

else-group:
`# else` *new-line group*$_{opt}$

endif-line:
`# endif` *new-line*

139) Thus, preprocessing directives are commonly called "lines." These "lines" have no other syntactic significance, as all white space is equivalent except in certain situations during preprocessing (see the `#` character string literal creation operator in 16.3.2, for example).

control-line:
```
# include      pp-tokens new-line
# define       identifier replacement-list new-line
# define       identifier lparen identifier-list_opt ) replacement-list new-line
# define       identifier lparen ... ) replacement-list new-line
# define       identifier lparen identifier-list, ... ) replacement-list new-line
# undef        identifier new-line
# line         pp-tokens new-line
# error        pp-tokens_opt new-line
# pragma       pp-tokens_opt new-line
#  new-line
```

text-line:
pp-tokens$_{opt}$ *new-line*

non-directive:
pp-tokens$_{opt}$ *new-line*

lparen:
a `(` character not immediately preceded by white-space

identifier-list:
identifier
identifier-list `,` *identifier*

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

2 A text line shall not begin with a # preprocessing token. A non-directive shall not begin with any of the directive names appearing in the syntax.

3 When in a group that is skipped (16.1), the directive syntax is relaxed to allow any sequence of preprocessing tokens to occur between the directive name and the following new-line character.

4 The only white-space characters that shall appear between preprocessing tokens within a preprocessing directive (from just after the introducing # preprocessing token through just before the terminating new-line character) are space and horizontal-tab (including spaces that have replaced comments or possibly other white-space characters in translation phase 3).

5 The implementation can process and skip sections of source files conditionally, include other source files, and replace macros. These capabilities are called *preprocessing*, because conceptually they occur before translation of the resulting translation unit.

6 The preprocessing tokens within a preprocessing directive are not subject to macro expansion unless otherwise stated.

[*Example:* In:

```
#define EMPTY
EMPTY   #   include <file.h>
```

the sequence of preprocessing tokens on the second line is *not* a preprocessing directive, because it does not begin with a # at the start of translation phase 4, even though it will do so after the macro `EMTPY` has been replaced. — *end example*]

16.1 Conditional inclusion [cpp.cond]

1 The expression that controls conditional inclusion shall be an integral constant expression except that: it shall not contain a cast; identifiers (including those lexically identical to keywords) are interpreted as described below;[140] and it may contain unary operator expressions of the form

`defined` *identifier*

or

`defined (` *identifier* `)`

which evaluate to `1` if the identifier is currently defined as a macro name (that is, if it is predefined or if it has been the subject of a `#define` preprocessing directive without an intervening `#undef` directive with the same subject identifier), `0` if it is not.

2 Each preprocessing token that remains after all macro replacements have occurred shall be in the lexical form of a token (2.6).

3 Preprocessing directives of the forms

`# if` *constant-expression new-line group*$_{opt}$
`# elif` *constant-expression new-line group*$_{opt}$

check whether the controlling constant expression evaluates to nonzero.

4 Prior to evaluation, macro invocations in the list of preprocessing tokens that will become the controlling constant expression are replaced (except for those macro names modified by the `defined` unary operator), just as in normal text. If the token `defined` is generated as a result of this replacement process or use of the `defined` unary operator does not match one of the two specified forms prior to macro replacement, the behavior is undefined. After all replacements due to macro expansion and the `defined` unary operator have been performed, all remaining identifiers and keywords[141], except for `true` and `false`, are replaced with the pp-number `0`, and then each preprocessing token is converted into a token. The resulting tokens comprise the controlling constant expression which is evaluated according to the rules of 5.19 using arithmetic that has at least the ranges specified in 18.2, except that all signed and unsigned integer types act as if they have the same representation as, respectively, `intmax_t` or `uintmax_t` (18.3.2). This includes interpreting character literals, which may involve converting escape sequences into execution character set members. Whether the numeric value for these character literals matches the value obtained when an identical character literal occurs in an expression (other than within a `#if` or `#elif` directive) is implementation-defined.[142] Also, whether a single-character character literal may have a negative value is implementation-defined. Each subexpression with type `bool` is subjected to integral promotion before processing continues.

5 Preprocessing directives of the forms

---

140) Because the controlling constant expression is evaluated during translation phase 4, all identifiers either are or are not macro names — there simply are no keywords, enumeration constants, and so on.

141) An alternative token (2.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not subject to this replacement.

142) Thus, the constant expression in the following `#if` directive and `if` statement is not guaranteed to evaluate to the same value in these two contexts.

```
#if 'z' - 'a' == 25
if ('z' - 'a' == 25)
```

`# ifdef` *identifier new-line group*$_{opt}$
`# ifndef` *identifier new-line group*$_{opt}$

check whether the identifier is or is not currently defined as a macro name. Their conditions are equivalent to `#if defined` *identifier* and `#if !defined` *identifier* respectively.

6 Each directive's condition is checked in order. If it evaluates to false (zero), the group that it controls is skipped: directives are processed only through the name that determines the directive in order to keep track of the level of nested conditionals; the rest of the directives' preprocessing tokens are ignored, as are the other preprocessing tokens in the group. Only the first group whose control condition evaluates to true (nonzero) is processed. If none of the conditions evaluates to true, and there is a `#else` directive, the group controlled by the `#else` is processed; lacking a `#else` directive, all the groups until the `#endif` are skipped.[143]

16.2 Source file inclusion [cpp.include]

1 A `#include` directive shall identify a header or source file that can be processed by the implementation.

2 A preprocessing directive of the form

`# include <` *h-char-sequence*`>` *new-line*

searches a sequence of implementation-defined places for a header identified uniquely by the specified sequence between the `<` and `>` delimiters, and causes the replacement of that directive by the entire contents of the header. How the places are specified or the header identified is implementation-defined.

3 A preprocessing directive of the form

`# include "` *q-char-sequence*`"` *new-line*

causes the replacement of that directive by the entire contents of the source file identified by the specified sequence between the `"` delimiters. The named source file is searched for in an implementation-defined manner. If this search is not supported, or if the search fails, the directive is reprocessed as if it read

`# include <` *h-char-sequence*`>` *new-line*

with the identical contained sequence (including `>` characters, if any) from the original directive.

4 A preprocessing directive of the form

`# include` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `include` in the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined.[144] The method by which a sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair or a pair of `"` characters is combined into a single header name preprocessing token is implementation-defined.

5 The implementation provides unique mappings for sequences consisting of one or more *nondigit*s or *digit*s (2.10) followed by a period (`.`) and a single *nondigit*. The first character shall not be a *digit*. The implementation may ignore the distinctions of alphabetical case.

6 A `#include` preprocessing directive may appear in a source file that has been read because of a `#include` directive in another file, up to an implementation-defined nesting limit.

143) As indicated by the syntax, a preprocessing token shall not follow a `#else` or `#endif` directive before the terminating new-line character. However, comments may appear anywhere in a source file, including within a preprocessing directive.
144) Note that adjacent string literals are not concatenated into a single string literal (see the translation phases in 2.1); thus, an expansion that results in two string literals is an invalid directive.

7 [*Note:* Although an implementation may provide a mechanism for making arbitrary source files available to the `< >` search, in general programmers should use the `< >` form for headers provided with the implementation, and the `" "` form for sources outside the control of the implementation. For instance:

```
#include <stdio.h>
#include <unistd.h>
#include "usefullib.h"
#include "myprog.h"
```

— *end note*]

[*Example:* Here is a macro-replaced `#include` directive:

```
#if VERSION == 1
    #define INCFILE  "vers1.h"
#elif VERSION == 2
    #define INCFILE  "vers2.h"   // and so on
#else
    #define INCFILE  "versN.h"
#endif
#include INCFILE
```

— *end example*]

16.3 Macro replacement [cpp.replace]

1 Two replacement lists are identical if and only if the preprocessing tokens in both have the same number, ordering, spelling, and white-space separation, where all white-space separations are considered identical.

2 An identifier currently defined as an *object-like* macro may be redefined by another `#define` preprocessing directive provided that the second definition is an object-like macro definition and the two replacement lists are identical, otherwise the program is ill-formed. Likewise, an identifier currently defined as a *function-like* macro may be redefined by another `#define` preprocessing directive provided that the second definition is a function-like macro definition that has the same number and spelling of parameters, and the two replacement lists are identical, otherwise the program is ill-formed.

3 There shall be white-space between the identifier and the replacement list in the definition of an object-like macro.

4 If the identifier-list in the macro definition does not end with an ellipsis, the number of arguments (including those arguments consisting of no preprocessing tokens) in an invocation of a function-like macro shall equal the number of parameters in the macro definition. Otherwise, there shall be more arguments in the invocation than there are parameters in the macro definition (excluding the `...`). There shall exist a `)` preprocessing token that terminates the invocation.

5 The identifier `__VA_ARGS__` shall occur only in the replacement-list of a function-like macro that uses the ellipsis notation in the parameters.

6 A parameter identifier in a function-like macro shall be uniquely declared within its scope.

7 The identifier immediately following the `define` is called the *macro name*. There is one name space for macro names. Any white-space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for either form of macro.

8 If a # preprocessing token, followed by an identifier, occurs lexically at the point at which a preprocessing directive could begin, the identifier is not subject to macro replacement.

9 A preprocessing directive of the form

```
# define identifier replacement-list new-line
```

defines an *object-like macro* that causes each subsequent instance of the macro name[145] to be replaced by the replacement list of preprocessing tokens that constitute the remainder of the directive.[146] The replacement list is then rescanned for more macro names as specified below.

10 A preprocessing directive of the form

```
# define identifier lparen identifier-list_opt ) replacement-list new-line
# define identifier lparen ... ) replacement-list new-line
# define identifier lparen identifier-list , ... ) replacement-list new-line
```

defines a *function-like macro* with parameters, similar syntactically to a function call. The parameters are specified by the optional list of identifiers, whose scope extends from their declaration in the identifier list until the new-line character that terminates the `#define` preprocessing directive. Each subsequent instance of the function-like macro name followed by a `(` as the next preprocessing token introduces the sequence of preprocessing tokens that is replaced by the replacement list in the definition (an invocation of the macro). The replaced sequence of preprocessing tokens is terminated by the matching `)` preprocessing token, skipping intervening matched pairs of left and right parenthesis preprocessing tokens. Within the sequence of preprocessing tokens making up an invocation of a function-like macro, new-line is considered a normal white-space character.

11 The sequence of preprocessing tokens bounded by the outside-most matching parentheses forms the list of arguments for the function-like macro. The individual arguments within the list are separated by comma preprocessing tokens, but comma preprocessing tokens between matching inner parentheses do not separate arguments. If (before argument substitution) any argument consists of no preprocessing tokens, the behavior is undefined. If there are sequences of preprocessing tokens within the list of arguments that would otherwise act as preprocessing directives, the behavior is undefined.

12 If there is a `...` in the identifier-list in the macro definition, then the trailing arguments, including any separating comma preprocessing tokens, are merged to form a single item: the variable arguments. The number of arguments so combined is such that, following merger, the number of arguments is one more than the number of parameters in the macro definition (excluding the `...`).

16.3.1 Argument substitution [cpp.subst]

1 After the arguments for the invocation of a function-like macro have been identified, argument substitution takes place. A parameter in the replacement list, unless preceded by a # or ## preprocessing token or followed by a ## preprocessing token (see below), is replaced by the corresponding argument after all macros contained therein have been expanded. Before being substituted, each argument's preprocessing tokens are completely macro replaced as if they formed the rest of the preprocessing file; no other preprocessing tokens are available.

2 An identifier `__VA_ARGS__` that occurs in the replacement list shall be treated as if it were a paremeter, and the variable arguments shall form the preprocessing tokens used to replace it.

16.3.2 The # operator [cpp.stringize]

1 Each # preprocessing token in the replacement list for a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement list.

145) Since, by macro-replacement time, all character literals and string literals are preprocessing tokens, not sequences possibly containing identifier-like subsequences (see 2.1, translation phases), they are never scanned for macro names or parameters.
146) An alternative token (2.5) is not an identifier, even when its spelling consists entirely of letters and underscores. Therefore it is not possible to define a macro whose name is the same as that of an alternative token.

2 If, in the replacement list, a parameter is immediately preceded by a # preprocessing token, both are replaced by a single character string literal preprocessing token that contains the spelling of the preprocessing token sequence for the corresponding argument. Each occurrence of white space between the argument's preprocessing tokens becomes a single space character in the character string literal. White space before the first preprocessing token and after the last preprocessing token comprising the argument is deleted. Otherwise, the original spelling of each preprocessing token in the argument is retained in the character string literal, except for special handling for producing the spelling of string literals and character literals: a \ character is inserted before each " and \ character of a character literal or string literal (including the delimiting " characters). If the replacement that results is not a valid character string literal, the behavior is undefined. The character string literal corresponding to an empty argument is "". The order of evaluation of # and ## operators is unspecified.

16.3.3 The ## operator [cpp.concat]

1 A ## preprocessing token shall not occur at the beginning or at the end of a replacement list for either form of macro definition.

2 If, in the replacement list of a function-like macro, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence; however, if an argument consists of no preprocessing tokens, the parameter is replaced by a placemarker preprocessing token instead.[147]

3 For both object-like and function-like macro invocations, before the replacement list is reexamined for more macro names to replace, each instance of a ## preprocessing token in the replacement list (not from an argument) is deleted and the preceding preprocessing token is concatenated with the following preprocessing token. Placemarker preprocessing tokens are handled specially; concatenation of two placemarkers results in a single placemarker preprocessing token, and concatenation of a placemarker with a non-placemarker preprocessing token results in the non-placemarker preprocessing token. If the result is not a valid preprocessing token, the behavior is undefined. The resulting token is available for further macro replacement. The order of evaluation of ## operators is unspecified.

[*Example:* In the following fragment:

```
#define hash_hash # ## #
#define mkstr(a) # a
#define in_between(a) mkstr(a)
#define join(c, d) in_between(c hash_hash d)
char p[] = join(x, y);          // equivalent to
                                // char p[] = "x ## y";
```

The expansion produces, at various stages:

```
join(x, y)
in_between(x hash_hash y)
in_between(x ## y)
mkstr(x ## y)
"x ## y"
```

In other words, expanding `hash_hash` produces a new token, consisting of two adjacent sharp signs, but this new token is not the ## operator. — *end example*]

16.3.4 Rescanning and further replacement [cpp.rescan]

1 After all parameters in the replacement list have been substituted and # and ## processing has taken

147) Placemarker preprocessing tokens do not appear in the syntax because they are temporary entities that exist only within translation phase 4.

place, all placemarker preprocessing tokens are removed. Then the resulting preprocessing token sequence is rescanned, along with all subsequent preprocessing tokens of the source file, for more macro names to replace.

2 If the name of the macro being replaced is found during this scan of the replacement list (not including the rest of the source file's preprocessing tokens), it is not replaced. Furthermore, if any nested replacements encounter the name of the macro being replaced, it is not replaced. These nonreplaced macro name preprocessing tokens are no longer available for further replacement even if they are later (re)examined in contexts in which that macro name preprocessing token would otherwise have been replaced.

3 The resulting completely macro-replaced preprocessing token sequence is not processed as a preprocessing directive even if it resembles one, but all pragma unary operator expressions within it are then processed as specified in 16.9 below.

16.3.5 Scope of macro definitions [cpp.scope]

1 A macro definition lasts (independent of block structure) until a corresponding `#undef` directive is encountered or (if none is encountered) until the end of the translation unit. Macro definitions have no significance after translation phase 4.

2 A preprocessing directive of the form

`# undef` *identifier new-line*

causes the specified identifier no longer to be defined as a macro name. It is ignored if the specified identifier is not currently defined as a macro name.

3 [*Note:* The simplest use of this facility is to define a "manifest constant," as in

```
#define TABSIZE 100
int table[TABSIZE];
```

4 The following defines a function-like macro whose value is the maximum of its arguments. It has the advantages of working for any compatible types of the arguments and of generating in-line code without the overhead of function calling. It has the disadvantages of evaluating one or the other of its arguments a second time (including side effects) and generating more code than a function if invoked several times. It also cannot have its address taken, as it has none.

```
#define max(a, b) ((a) > (b) ? (a) : (b))
```

The parentheses ensure that the arguments and the resulting expression are bound properly.

5 To illustrate the rules for redefinition and reexamination, the sequence

```
#define x       3
#define f(a)    f(x * (a))
#undef  x
#define x       2
#define g       f
#define z       z[0]
#define h       g(~
#define m(a)    a(w)
#define w       0,1
#define t(a)    a
#define p()     int
#define q(x)    x
#define r(x,y)  x ## y
#define str(x)  # x
```

```
f(y+1) + f(f(z)) % t(t(g)(0) + t)(1);
g(x+(3,4)-w) | h 5) & m
    (f)^m(m);
p() i[q()] = { q(1), r(2,3), r(4,), r(,5), r(,) };
char c[2][6] = { str(hello), str() };
```

results in

```
f(2 * (y+1)) + f(2 * (f(2 * (z[0])))) % f(2 * (0)) + t(1);
f(2 * (2+(3,4)-0,1)) | f(2 * (~5)) & f(2 * (0,1))^m(0,1);
int i[] = { 1, 23, 4, 5, };
char c[2][6] = { "hello", "" };
```

6 To illustrate the rules for creating character string literals and concatenating tokens, the sequence

```
#define str(s)      # s
#define xstr(s)     str(s)
#define debug(s, t) printf("x" # s "= %d, x" # t "= %s", \
                x ## s, x ## t)
#define INCFILE(n)  vers ## n  /* from previous #include example */
#define glue(a, b)  a ## b
#define xglue(a, b) glue(a, b)
#define HIGHLOW     "hello"
#define LOW         LOW ", world"

debug(1, 2);
fputs(str(strncmp("abc\0d", "abc", '\4')  // this goes away
    == 0) str(: @\n), s);
#include xstr(INCFILE(2).h)
glue(HIGH, LOW);
xglue(HIGH, LOW)
```

results in

```
printf("x" "1" "= %d, x" "2" "= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0" ": @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello" ", world"
```

or, after concatenation of the character string literals,

```
printf("x1= %d, x2= %s", x1, x2);
fputs("strncmp(\"abc\\0d\", \"abc\", '\\4') == 0: @\n", s);
#include "vers2.h"    (after macro replacement, before file access)
"hello";
"hello, world"
```

Space around the # and ## tokens in the macro definition is optional.

7 To illustrate the rules for placemarker preprocessing tokens, the sequence

```
#define t(x,y.z) x ## y ## z
int j[] = { t(1,2,3), t(,4,5), t(6,{,}7), t(8,9,),
  t(10,{,}), t(,11,), t(,{,}12), t(,{,}) };
```

results in

```
int j[] = { 123, 45, 67, 89,
  10, 11, 12, };
```

8 To demonstrate the redefinition rules, the following sequence is valid.

```
#define OBJ_LIKE      (1-1)
#define OBJ_LIKE      /* white space */ (1-1) /* other */
#define FTN_LIKE(a)   ( a )
#define FTN_LIKE( a )(      /* note the white space */ \
                a /* other stuff on this  line
                */ )
```

9 But the following redefinitions are invalid:

```
#define OBJ_LIKE    (0)      // different token sequence
#define OBJ_LIKE    (1 - 1)  // different white space
#define FTN_LIKE(b) ( a )    // different parameter usage
#define FTN_LIKE(b) ( b )    // different parameter spelling
```

— *end note*]

10 Finally, to show the variable argument list macro facilities:

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test) ? puts(#test) : printf(__VA_ARGS__))
debug("Flag");
debug("X = %d\n", x);
showlist(The first, second, and third items.);
report(x>y, "x is %d but y is \%d", x, y);
```

results in

```
fprintf(stderr, "Flag" );
fprintf(stderr, "X = %d\n", x );
puts( "The first, second, and third items." );
((x>y) ? puts("x>y") : printf("x is %d but y is %d", x, y));
```

16.4 Line control [cpp.line]

1 The string literal of a `#line` directive, if present, shall be a character string literal.

2 The *line number* of the current source line is one greater than the number of new-line characters read or introduced in translation phase 1 (2.1) while processing the source file to the current token.

3 A preprocessing directive of the form

`# line` *digit-sequence new-line*

causes the implementation to behave as if the following sequence of source lines begins with a source line that has a line number as specified by the digit sequence (interpreted as a decimal integer). If the digit sequence specifies zero or a number greater than 2147483647, the behavior is undefined.

4 A preprocessing directive of the form

`# line` *digit-sequence* `"` *s-char-sequence*$_{opt}$ `"` *new-line*

sets the line number similarly and changes the presumed name of the source file to be the contents of the character string literal.

5 A preprocessing directive of the form

`# line` *pp-tokens new-line*

(that does not match one of the two previous forms) is permitted. The preprocessing tokens after `line` on the directive are processed just as in normal text (each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens). If the directive resulting after all replacements does not match one of the two previous forms, the behavior is undefined; otherwise, the result is processed as appropriate.

16.5 Error directive [cpp.error]

1 A preprocessing directive of the form

`# error` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to produce a diagnostic message that includes the specified sequence of preprocessing tokens, and renders the program ill-formed.

16.6 Pragma directive [cpp.pragma]

1 A preprocessing directive of the form

`# pragma` *pp-tokens*$_{opt}$ *new-line*

causes the implementation to behave in an implementation-defined manner. The behavior might cause translation to fail or cause the translator or the resulting program to behave in a non-conforming manner. Any pragma that is not recognized by the implementation is ignored.

16.7 Null directive [cpp.null]

1 A preprocessing directive of the form

`#` *new-line*

has no effect.

16.8 Predefined macro names [cpp.predefined]

1 The following macro names shall be defined by the implementation:

`__cplusplus`
The name `__cplusplus` is defined to the value `[tbd]` when compiling a C++ translation unit.[148]

`__DATE__`
The date of translation of the source file (a character string literal of the form `"Mmm dd yyyy"`, where the names of the months are the same as those generated by the `asctime` function, and the first character of `dd` is a space character if the value is less than 10). If the date of translation is not available, an implementation-defined valid date is supplied.

`__FILE__`
The presumed name of the source file (a character string literal).

148) It is intended that future versions of this standard will replace the value of this macro with a greater value. Non-conforming compilers should use a value with at most five decimal digits.

`__LINE__`
The line number of the current source line (a decimal constant).

`__STDC_HOSTED__`
The integer constant `1` if the implementation is a hosted implementation or the integer constant `0` if it is not.

`__TIME__`
The time of translation of the source file (a character string literal of the form `"hh:mm:ss"` as in the time generated by the `asctime` function). If the time of translation is not available, an implementation-defined valid time is supplied.

2 The following macro names are conditionally defined by the implementation:

`__STDC__`
Whether `__STDC__` is predefined and if so, what its value is, are implementation-defined.

`__STDC_VERSION__`
Whether `__STDC_VERSION__` is predefined and if so, what its value is, are implementation-defined.

`__STDC_ISO_10646__`
An integer constant of the form `yyyymmL` (for example, `199712L`), intended to indicate that values of type `wchar_t` are the coded representations of the characters defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month.

3 The values of the predefined macros (except for `__LINE__` and `__FILE__`) remain constant throughout the translation unit.

4 If any of the pre-defined macro names in this subclause, or the identifier `defined`, is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is undefined. Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.

16.9 Pragma operator [cpp.pragma.op]

A unary operator expression of the form:

`_Pragma (` *string-literal* `)`

is processed as follows: The string literal is *destringized* by deleting the `L` prefix, if present, deleting the leading and trailing double-quotes, replacing each escape sequence `\"` by a double-quote, and replacing each escape sequence `\\` by a single backslash. The resulting sequence of characters is processed through translation phase 3 to produce preprocessing tokens that are executed as if they were the *pp-tokens* in a pragma directive. The original four preprocessing tokens in the unary operator expression are removed.

[*Example:*

```
#pragma listing on "..\listing.dir"
```

can also be expressed as:

```
_Pragma ( listing on "\"..\\listing.dir\"" )
```

The latter form is processed in the same way whether it appears literally as shown, or results from macro replacement, as in:

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING( ..\listing.dir )
```

— *end example*]

17 Library introduction [library]

17.1 General [library.general]

1 This Clause describes the contents of the *C++ standard library*, how a well-formed C++ program makes use of the library, and how a conforming implementation may provide the entities in the library.

2 The C++ standard library provides an extensible framework, and contains components for: language support, diagnostics, general utilities, strings, locales, containers, iterators, algorithms, numerics, and input/output. The language support components are required by certain parts of the C++ language, such as memory allocation (5.3.4, 5.3.5) and exception processing (Clause 15).

3 The general utilities include components used by other library elements, such as a predefined storage allocator for dynamic storage management (3.7.4). The diagnostics components provide a consistent framework for reporting errors in a C++ program, including predefined exception classes.

4 The strings components provide support for manipulating text represented as sequences of type `char`, sequences of type `char16_t`, sequences of type `char32_t`, sequences of type `wchar_t`, and sequences of any other character-like type. The localization components extend internationalization support for such text processing.

5 The containers, iterators, and algorithms provide a C++ program with access to a subset of the most widely used algorithms and data structures.

6 Numeric algorithms and the complex number components extend support for numeric processing. The `valarray` components provide support for *n*-at-a-time processing, potentially implemented as parallel operations on platforms that support such processing.

7 The `iostream` components are the primary mechanism for C++ program input/output. They can be used with other elements of the library, particularly strings, locales, and iterators.

8 The atomic components allow more fine-grained concurrent access to shared data than is possible with locks.

9 This library also makes available the facilities of the C99 standard library, suitably adjusted to ensure static type safety.

10 The descriptions of many library functions rely on the Standard C99 Library for the signatures and semantics of those functions. In all such cases, any use of the `restrict` qualifier shall be omitted.

17.2 Overview [library.overview]

1 The following subclauses describe the definitions (17.3), method of description (17.5), and organization (17.6.2) of the library. Clause 17.6, Clauses 18 through 30, and Annex D specify the contents of the library, as well as library requirements and constraints on both well-formed C++ programs and conforming implementations.

2 Detailed specifications for each of the components in the library are in Clauses 18–30, as shown in Table 12.

17.3 Definitions [definitions]

17.3.1 [defns.arbitrary.stream]

arbitrary-positional stream

Table 12 — Library categories

Clause	Category
18	Language support
19	Diagnostics
20	General utilities
21	Strings
22	Localization
23	Containers
24	Iterators
25	Algorithms
26	Numerics
27	Input/output
28	Regular expressions
29	Atomic operations
30	Threads

a stream (described in Clause 27) that can seek to any integral position within the length of the stream. Every arbitrary-positional stream is also a repositional stream ().

17.3.2 [defns.blocked]

blocked thread

a thread that is waiting for some condition (other than the availability of a processor) to be satisfied before it can continue execution.[149] As a verb, to block is to place a thread in the blocked state, and to unblock is to place a thread in the unblocked state.

17.3.3 [defns.character]

character

in Clauses 21, 22, and 27, means any object which, when treated sequentially, can represent text. The term does not only mean `char`, `char16_t`, `char32_t`, and `wchar_t` objects, but any value that can be represented by a type that provides the definitions specified in these Clauses.

17.3.4 [defns.character.container]

character container type

a class or a type used to represent a *character*. It is used for one of the template parameters of the string and iostream class templates. A character container type shall be a POD (3.9) type.

17.3.5 [defns.comparison]

comparison function

an operator function (13.5) for any of the equality (5.10) or relational (5.9) operators.

17.3.6 [defns.component]

component

a group of library entities directly related as members, parameters, or return types. For example, the class template `basic_string` and the non-member function templates that operate on strings are referred to as the *string component*.

149) This definition is taken from POSIX.

17.3.7 **[defns.deadlock]**

deadlock

two or more threads are unable to continue execution because each is blocked waiting for one or more of the others to satisfy some condition.

17.3.8 **[defns.default.behavior]**

default behavior

a description of *replacement function* and *handler function* semantics. Any specific behavior provided by the implementation, within the scope of the *required behavior*.

17.3.9 **[defns.handler]**

handler function

a *non-reserved function* whose definition may be provided by a C++ program. A C++ program may designate a handler function at various points in its execution, by supplying a pointer to the function when calling any of the library functions that install handler functions (Clause 18).

17.3.10 **[defns.iostream.templates]**

iostream class templates

templates, defined in Clause 27, that take two template arguments: `charT` and `traits`. The argument `charT` is a character container class, and the argument `traits` is a class which defines additional characteristics and functions of the character type represented by `charT` necessary to implement the iostream class templates.

17.3.11 **[defns.modifier]**

modifier function

a class member function (9.3), other than constructors, assignment, or destructor, that alters the state of an object of the class.

17.3.12 **[defns.obj.state]**

object state

the current value of all non-static class members of an object (9.2). The state of an object can be obtained by using one or more *observer functions*.

17.3.13 **[defns.ntcts]**

NTCTS

a sequence of values that have *character type*, that precede the terminating null character type value `charT()`.

17.3.14 **[defns.narrow.iostream]**

narrow-oriented iostream classes

the instantiations of the iostream class templates on the character container class `char` and the default value of any other parameters. The traditional iostream classes are regarded as the narrow-oriented iostream classes (27.3.1).

17.3.15 **[defns.observer]**

observer function

a class member function (9.3) that accesses the state of an object of the class, but does not alter that state. Observer functions are specified as `const` member functions (9.3.2).

17.3.16 **[defns.replacement]**

replacement function

a *non-reserved function* whose definition is provided by a C++ program. Only one definition for such a function is in effect for the duration of the program's execution, as the result of creating the program (2.1) and resolving the definitions of all translation units (3.5).

17.3.17 **[defns.repositional.stream]**

repositional stream

a stream (described in Clause 27) that can seek only to a position that was previously encountered.

17.3.18 **[defns.required.behavior]**

required behavior

a description of *replacement function* and *handler function* semantics, applicable to both the behavior provided by the implementation and the behavior that shall be provided by any function definition in the program. If a function defined in a C++ program fails to meet the required behavior when it executes, the behavior is undefined.

17.3.19 **[defns.reserved.function]**

reserved function

a function, specified as part of the C++ standard library, that must be defined by the implementation. If a C++ program provides a definition for any reserved function, the results are undefined.

17.3.20 **[defns.stable]**

stable algorithm

an algorithm that preserves, as appropriate to the particular algorithm, the order of elements.

— For the *sort* algorithms the relative order of equivalent elements is preserved.

— For the *remove* algorithms the relative order of the elements that are not removed is preserved.

— For the *merge* algorithms, for equivalent elements in the original two ranges, the elements from the first range precede the elements from the second range.

17.3.21 **[defns.traits]**

traits class

a class that encapsulates a set of types and functions necessary for class templates and function templates to manipulate objects of types for which they are instantiated. Traits classes defined in Clauses 21, 22 and 27 are *character traits*, which provide the character handling support needed by the string and iostream classes.

17.3.22 **[defns.wide.iostream]**

wide-oriented iostream classes

the instantiations of the iostream class templates on the character container class `wchar_t` and the default value of any other parameters.

17.4 Additional definitions [defns.additional]

1 1.3 defines additional terms used elsewhere in this International Standard.

17.5 Method of description (Informative) [description]

17.5.1 General [description.general]

1 This subclause describes the conventions used to specify the C++ standard library. 17.5.2 describes the structure of the normative Clauses 18 through 30 and Annex D. 17.5.3 describes other editorial conventions.

17.5.2 Structure of each clause [structure]

17.5.2.1 Elements [structure.elements]

1 Each library clause contains the following elements, as applicable:[150]

- Summary
- Requirements
- Detailed specifications
- References to the Standard C library

17.5.2.2 Summary [structure.summary]

1 The Summary provides a synopsis of the category, and introduces the first-level subclauses. Each subclause also provides a summary, listing the headers specified in the subclause and the library entities provided in each header.

2 Paragraphs labelled "Note(s):" or "Example(s):" are informative, other paragraphs are normative.

3 The summary and the detailed specifications are presented in the order:

- macros
- values
- types
- classes
- functions
- objects

17.5.2.3 Requirements [structure.requirements]

1 Requirements describe contraints that shall be met by a C++ program that extends the standard library. Such extensions are generally one of the following:

- Template arguments
- Derived classes
- Containers, iterators, and algorithms that meet an interface convention

150) To save space, items that do not apply to a Clause are omitted. For example, if a Clause does not specify any requirements, there will be no "Requirements" subclause.

2 The string and iostream components use an explicit representation of operations required of template arguments. They use a class template `char_traits` to define these constraints.

3 Interface convention requirements are stated as generally as possible. Instead of stating "class X has to define a member function `operator++()`," the interface requires "for any object `x` of class `X`, `++x` is defined." That is, whether the operator is a member is unspecified.

4 Requirements are stated in terms of well-defined expressions that define valid terms of the types that satisfy the requirements, or concepts that define capabilities of the types that satisfy the requirements. For every set of well-defined expression requirements there is a table that specifies an initial set of the valid expressions and their semantics. For every set of concept requirements there is a concept that specifies the requirements and their semantics (20.1, 23.1.6, 24.1). Any generic algorithm (Clause 25) that uses the well-defined expression requirements is described in terms of the valid expressions for its formal type parameters. Any generic algorithm that uses concepts places requirements on its formal type parameters.

5 Template argument requirements are sometimes referenced by name. See 17.5.3.2.

6 In some cases the semantic requirements are presented as C++ code. Such code is intended as a specification of equivalence of a construct to another construct, not necessarily as the way the construct must be implemented.[151]

17.5.2.4 Detailed Specifications [structure.specifications]

1 The detailed specifications each contain the following elements:[152]

- name and brief description
- synopsis (class definition or function prototype, as appropriate)
- restrictions on template arguments, if any
- description of class invariants
- description of function semantics

2 Descriptions of class member functions follow the order (as appropriate):[153]

- constructor(s) and destructor
- copying & assignment functions
- comparison functions
- modifier functions
- observer functions
- operators and other non-member functions

3 Descriptions of function semantics contain the following elements (as appropriate):[154]

- *Requires:* the preconditions for calling the function
- *Effects:* the actions performed by the function

151) Although in some cases the code given is unambiguously the optimum implementation.
152) The form of these specifications was designed to follow the conventions established by existing C++ library vendors.
153) To save space, items that do not apply to a class are omitted. For example, if a class does not specify any comparison functions, there will be no "Comparison functions" subclause.
154) To save space, items that do not apply to a function are omitted. For example, if a function does not specify any further preconditions, there will be no "Requires" paragraph.

— *Postconditions:* the observable results established by the function

— *Returns:* a description of the value(s) returned by the function

— *Throws:* any exceptions thrown by the function, and the conditions that would cause the exception

— *Complexity:* the time and/or space complexity of the function

— *Remarks:* additional semantic constraints on the function

— *Error conditions:* the error conditions for error codes reported by the function.

— *Notes:* non-normative comments about the function

4 For non-reserved replacement and handler functions, Clause 18 specifies two behaviors for the functions in question: their required and default behavior. The *default behavior* describes a function definition provided by the implementation. The *required behavior* describes the semantics of a function definition provided by either the implementation or a C++ program. Where no distinction is explicitly made in the description, the behavior described is the required behavior.

5 Complexity requirements specified in the library Clauses are upper bounds, and implementations that provide better complexity guarantees satisfy the requirements.

6 Error conditions specify conditions where a function may fail. The conditions are listed, together with a suitable explanation, as the `enum class errc` constants (19.4) that could be used as an argument to function `make_error_condition` (19.4.3.6).

17.5.2.5 C Library [structure.see.also]

1 Paragraphs labelled "See also:" contain cross-references to the relevant portions of this International Standard and the ISO C standard, which is incorporated into this International Standard by reference.

17.5.3 Other conventions [conventions]

17.5.3.1 General [conventions.general]

1 This subclause describes several editorial conventions used to describe the contents of the C++ standard library. These conventions are for describing implementation-defined types (17.5.3.2), and member functions (17.5.3.3).

17.5.3.2 Type descriptions [type.descriptions]

17.5.3.2.1 General [type.descriptions.general]

1 The Requirements subclauses may describe names that are used to specify constraints on template arguments.[155] These names are used in library Clauses to describe the types that may be supplied as arguments by a C++ program when instantiating template components from the library.

2 Certain types defined in Clause 27 are used to describe implementation-defined types. They are based on other types, but with added constraints.

17.5.3.2.2 Enumerated types [enumerated.types]

1 Several types defined in Clause 27 are *enumerated types*. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration.[156]

155) Examples from 20.1 include: `EqualityComparable`, `LessThanComparable`, `CopyConstructable`, etc. Examples from 24.1.1 include: `InputIterator`, `ForwardIterator`, `Function`, `Predicate`, etc.

156) Such as an integer type, with constant integer values (3.9.1).

2 The enumerated type *enumerated* can be written:

```
enum enumerated { V0, V1, V2, V3, .....};

static const enumerated C0 (V0);
static const enumerated C1 (V1);
static const enumerated C2 (V2);
static const enumerated C3 (V3);
  .....
```

3 Here, the names *C0*, *C1*, etc. represent *enumerated elements* for this particular enumerated type. All such elements have distinct values.

17.5.3.2.3 Bitmask types [bitmask.types]

1 Several types defined in Clauses 18 through 30 and Annex D are *bitmask types*. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a `bitset` (20.2.6).

2 The bitmask type *bitmask* can be written:

```
enum bitmask {
  V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, .....
};

static const bitmask C0(V0);
static const bitmask C1(V1);
static const bitmask C2(V2);
static const bitmask C3(V3);
  .....

// For exposition only.
// int_type is an integral type capable of
// representing all values of bitmask
bitmask operator& (bitmask X, bitmask Y) {
  return static_cast<bitmask>(
    static_cast<int_type>(X) &
    static_cast<int_type>(Y));
}
bitmask operator| (bitmask X, bitmask Y) {
  return static_cast<bitmask>(
    static_cast<int_type>(X) |
    static_cast<int_type>(Y));
}
bitmask operator^ (bitmask X, bitmask Y){
  return static_cast<bitmask>(
    static_cast<int_type>(X) ^
    static_cast<int_type>(Y));
}
bitmask operator~ (bitmask X){
  return static_cast<bitmask>(~static_cast<int_type>(X));
}
bitmask& operator&=(bitmask& X, bitmask Y){
  X = X&Y; return X;
}
bitmask& operator|=(bitmask& X, bitmask Y) {
  X = X|Y; return X;
```

```
}
bitmask& operator^=(bitmask& X, bitmask Y) {
  X = X^Y; return X;
}
```

3 Here, the names *C0*, *C1*, etc. represent *bitmask elements* for this particular bitmask type. All such elements have distinct values such that, for any pair *Ci* and *Cj*, *Ci* & *Ci* is nonzero and *Ci* & *Cj* is zero.

4 The following terms apply to objects and values of bitmask types:

— To *set* a value Y in an object X is to evaluate the expression $X \mid= Y$.

— To *clear* a value Y in an object X is to evaluate the expression $X \&= \sim Y$.

— The value Y *is set* in the object X if the expression $X \& Y$ is nonzero.

17.5.3.2.4 Character sequences [character.seq]

17.5.3.2.4.1 General [character.seq.general]

1 The C standard library makes widespread use of characters and character sequences that follow a few uniform conventions:

— A *letter* is any of the 26 lowercase or 26 uppercase letters in the basic execution character set.[157]

— The *decimal-point character* is the (single-byte) character used by functions that convert between a (single-byte) character sequence and a value of one of the floating-point types. It is used in the character sequence to denote the beginning of a fractional part. It is represented in Clauses 18 through 27 and Annex D by a period, `'.'`, which is also its value in the `"C"` locale, but may change during program execution by a call to `setlocale(int, const char*)`,[158] or by a change to a `locale` object, as described in Clauses 22.1 and 27.

— A *character sequence* is an array object (8.3.4) A that can be declared as *`T A [N ]`*, where T is any of the types `char`, `unsigned char`, or `signed char` (3.9.1), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A character sequence can be designated by a pointer value S that points to its first element.

17.5.3.2.4.2 Byte strings [byte.strings]

1 A *null-terminated byte string*, or NTBS, is a character sequence whose highest-addressed element with defined content has the value zero (the *terminating null* character).[159]

2 The *length* of an NTBS is the number of elements that precede the terminating null character. An *empty* NTBS has a length of zero.

3 The *value* of an NTBS is the sequence of values of the elements up to and including the terminating null character.

157) Note that this definition differs from the definition in ISO C 7.1.1.
158) declared in `<clocale>` (22.4).
159) Many of the objects manipulated by function signatures declared in `<cstring>` (21.5) are character sequences or NTBSs. The size of some of these character sequences is limited by a length value, maintained separately from the character sequence.

4 A *static* NTBS is an NTBS with static storage duration.[160]

17.5.3.2.4.3 Multibyte strings [multibyte.strings]

1 A *null-terminated multibyte string,* or NTMBS, is an NTBS that constitutes a sequence of valid multibyte characters, beginning and ending in the initial shift state.[161]

2 A *static* NTMBS is an NTMBS with static storage duration.

17.5.3.2.4.4 `char16_t` sequences [char16_t.seq]

1 A *char16-character sequence* is an array object (8.3.4) A that can be declared as `T A[N]`, where T is type `char16_t` (3.9.1), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A char16-character sequence can be designated by a pointer value S that designates its first element.

2 A *null-terminated char16-character string,* or NTC16S, is a char16-character sequence whose highest-addressed element with defined content has the value zero.[162]

3 The *length* of an NTC16S is the number of elements that precede the terminating null `char16_t` character. An *empty* NTC16S has a length of zero.

4 The *value* of an NTC16S is the sequence of values of the elements up to and including the terminating null character.

5 A *static* NTC16S is an NTC16S with static storage duration.[163]

17.5.3.2.4.5 `char32_t` sequences [char32_t.seq]

1 A *char32-character sequence* is an array object (8.3.4) A that can be declared as `T A[N]`, where T is type `char32_t` (3.9.1), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A char32-character sequence can be designated by a pointer value S that designates its first element.

2 A *null-terminated char32-character string,* or NTC32S, is a char32-character sequence whose highest-addressed element with defined content has the value zero.[164]

3 The *length* of an NTC32S is the number of elements that precede the terminating null `char32_t` character. An *empty* NTC32S has a length of zero.

4 The *value* of an NTC32S is the sequence of values of the elements up to and including the terminating null character.

5 A *static* NTC32S is an NTC32S with static storage duration.[165]

17.5.3.2.4.6 Wide-character sequences [wide.characters]

1 A *wide-character sequence* is an array object (8.3.4) A that can be declared as `T A[N]`, where T is type `wchar_t` (3.9.1), optionally qualified by any combination of `const` or `volatile`. The initial elements of the array have defined contents up to and including an element determined by some predicate. A wide-character sequence can be designated by a pointer value S that designates its first element.

160) A string literal, such as `"abc"`, is a static NTBS
161) An NTBS that contains characters only from the basic execution character set is also an NTMBS. Each multibyte character then consists of a single byte.
162) Many of the objects manipulated by function signatures declared in `<cuchar>` are char16-character sequences or NTC16Ss.
163) A `char16_t` string literal, such as `u"abc"`, is a static NTC16S.
164) Many of the objects manipulated by function signatures declared in `<cuchar>` are char32-character sequences or NTC32Ss.
165) A `char32_t` string literal, such as `U"abc"`, is a static NTC32S.

2 A *null-terminated wide-character string,* or NTWCS, is a wide-character sequence whose highest-addressed element with defined content has the value zero.[166]

3 The *length* of an NTWCS is the number of elements that precede the terminating null wide character. An *empty* NTWCS has a length of zero.

4 The *value* of an NTWCS is the sequence of values of the elements up to and including the terminating null character.

5 A *static* NTWCS is an NTWCS with static storage duration.[167]

17.5.3.3 Functions within classes [functions.within.classes]

1 For the sake of exposition, Clauses 18 through 27 and Annex D do not describe copy constructors, assignment operators, or (non-virtual) destructors with the same apparent semantics as those that can be generated by default (12.1, 12.4, 12.8).

2 It is unspecified whether the implementation provides explicit definitions for such member function signatures, or for virtual destructors that can be generated by default.

17.5.3.4 Private members [objects.within.classes]

1 Clauses 18 through 30 and Annex D do not specify the representation of classes, and intentionally omit specification of class members (9.2). An implementation may define static or non-static class members, or both, as needed to implement the semantics of the member functions specified in Clauses 18 through 30 and Annex D.

2 Objects of certain classes are sometimes required by the external specifications of their classes to store data, apparently in member objects. For the sake of exposition, some subclauses provide representative declarations, and semantic requirements, for private member objects of classes that meet the external specifications of the classes. The declarations for such member objects and the definitions of related member types are enclosed in a comment that ends with *exposition only*, as in:

```
// streambuf* sb;                                    exposition only
```

3 An implementation may use any technique that provides equivalent external behavior.

17.6 Library-wide requirements [requirements]

17.6.1 General [requirements.general]

1 This subclause specifies requirements that apply to the entire C++ standard library. Clauses 18 through 30 and Annex D specify the requirements of individual entities within the library.

2 Requirements specified in terms of interactions between threads do not apply to programs having only a single thread of execution.

3 Within this subclause, 17.6.2 describes the library's contents and organization, 17.6.3 describes how well-formed C++ programs gain access to library entities, 17.6.4 describes constraints on well-formed C++ programs, and 17.6.5 describes constraints on conforming implementations.

17.6.2 Library contents and organization [organization]

17.6.2.1 General [organization.general]

1 17.6.2.2 describes the entities defined in the C++ standard library. 17.6.2.3 lists the standard library headers

166) Many of the objects manipulated by function signatures declared in `<cwchar>` are wide-character sequences or NTWCSs.
167) A wide string literal, such as `L"abc"` is a static NTWCS.

and some constraints on those headers. 17.6.2.4 lists requirements for a freestanding implementation of the C++ standard library.

17.6.2.2 Library contents [contents]

1 The C++ standard library provides definitions for the following types of entities: macros, values, types, concepts, concept maps, templates, classes, functions, objects.

2 All library entities except macros, `operator new` and `operator delete` are defined within the namespace `std` or namespaces nested within namespace `std`.

3 Whenever a name `x` defined in the standard library is mentioned, the name `x` is assumed to be fully qualified as `::std::x`, unless explicitly described otherwise. For example, if the Effects section for library function `F` is described as calling library function `G`, the function `::std::G` is meant.

17.6.2.3 Headers [headers]

1 Each element of the C++ standard library is declared or defined (as appropriate) in a *header*.[168]

2 The C++ standard library provides 54 *C++ library headers*, as shown in Table 13.

Table 13 — C++ library headers

<algorithm>	<forward_list>	<iterator_concepts>	<queue>	<system_error>
<array>	<fstream>	<limits>	<random>	<threads>
<bitset>	<functional>	<list>	<ratio>	<tuple>
<chrono>	<future>	<locale>	<regex>	<typeinfo>
<codecvt>	<initializer_list>	<map>	<set>	<type_traits>
<complex>	<iomanip>	<memory>	<sstream>	<unordered_map>
<concepts>	<ios>	<memory_concepts>	<stack>	<unordered_set>
<condition_variable>	<iosfwd>	<mutex>	<stdexcept>	<utility>
<container_concepts>	<iostream>	<new>	<streambuf>	<valarray>
<deque>	<istream>	<numeric>	<string>	<vector>
<exception>	<iterator>	<ostream>	<strstream>	

3 The facilities of the C99 standard Library are provided in 26 additional headers, as shown in Table 14.

Table 14 — C++ headers for C library facilities

<cassert>	<cinttypes>	<csignal>	<cstdio>	<cwchar>
<ccomplex>	<ciso646>	<cstdarg>	<cstdlib>	<cwctype>
<cctype>	<climits>	<cstdatomic>	<cstring>	
<cerrno>	<clocale>	<cstdbool>	<ctgmath>	
<cfenv>	<cmath>	<cstddef>	<ctime>	
<cfloat>	<csetjmp>	<cstdint>	<cuchar>	

4 Except as noted in Clauses 18 through 30 and Annex D, the contents of each header `c`*`name`* shall be the same as that of the corresponding header *`name`*`.h`, as specified in the C99 standard Library (1.2) or the C Unicode TR, as appropriate, as if by inclusion. In the C++ standard library, however, the declarations (except for names which are defined as macros in C) are within namespace scope (3.3.5) of the namespace `std`. It is

168) A header is not necessarily a source file, nor are the sequences delimited by < and > in header names necessarily valid source file names (16.2).

unspecified whether these names are first declared within the global namespace scope and are then injected into namespace `std` by explicit *using-declaration*s (7.3.3).

5 Names which are defined as macros in C shall be defined as macros in the C++ standard library, even if C grants license for implementation as functions. [*Note:* the names defined as macros in C include the following: `assert`, `offsetof`, `setjmp`, `va_arg`, `va_end`, and `va_start`. — *end note*]

6 Names that are defined as functions in C shall be defined as functions in the C++ standard library.[169]

7 Identifiers that are keywords or operators in C++ shall not be defined as macros in C++ standard library headers.[170]

8 D.5, C standard library headers, describes the effects of using the *`name`*`.h` (C header) form in a C++ program.[171]

17.6.2.4 Freestanding implementations [compliance]

1 Two kinds of implementations are defined: *hosted* and *freestanding* (1.4). For a hosted implementation, this International Standard describes the set of available headers.

2 A freestanding implementation has an implementation-defined set of headers. This set shall include at least the headers shown in Table 15.

Table 15 — C++ headers for freestanding implementations

Subclause		Header(s)
18.1	Types	`<cstddef>`
18.2	Implementation properties	`<limits>`
18.4	Start and termination	`<cstdlib>`
18.5	Dynamic memory management	`<new>`
18.6	Type identification	`<typeinfo>`
18.7	Exception handling	`<exception>`
18.9	Other runtime support	`<cstdarg>`

3 The supplied version of the header `<cstdlib>` shall declare at least the functions `abort()`, `atexit()`, and `exit()` (18.4).

17.6.3 Using the library [using]

17.6.3.1 Overview [using.overview]

1 This section describes how a C++ program gains access to the facilities of the C++ standard library. 17.6.3.2 describes effects during translation phase 4, while 17.6.3.3 describes effects during phase 8 (2.1).

17.6.3.2 Headers [using.headers]

1 The entities in the C++ standard library are defined in headers, whose contents are made available to a translation unit when it contains the appropriate `#include` preprocessing directive (16.2).

169) This disallows the practice, allowed in C, of providing a masking macro in addition to the function prototype. The only way to achieve equivalent inline behavior in C++ is to provide a definition as an extern inline function.
170) In particular, including the standard header `<iso646.h>` or `<ciso646>` has no effect.
171) The `".h"` headers dump all their names into the global namespace, whereas the newer forms keep their names in namespace `std`. Therefore, the newer forms are the preferred forms for all uses except for C++ programs which are intended to be strictly compatible with C.

2 A translation unit may include library headers in any order (Clause 2). Each may be included more than once, with no effect different from being included exactly once, except that the effect of including either `<cassert>` or `<assert.h>` depends each time on the lexically current definition of `NDEBUG`.[172]

3 A translation unit shall include a header only outside of any external declaration or definition, and shall include the header lexically before the first reference to any of the entities it declares in that translation unit.

17.6.3.3 Linkage [using.linkage]

1 Entities in the C++ standard library have external linkage (3.5). Unless otherwise specified, objects and functions have the default `extern "C++"` linkage (7.5).

2 Whether a name from the Standard C library declared with external linkage has `extern "C"` or `extern "C++"` linkage is implementation-defined. It is recommended that an implementation use `extern "C++"` linkage for this purpose.[173]

3 Objects and functions defined in the library and required by a C++ program are included in the program prior to program startup.

See also: replacement functions (17.6.4.6), run-time changes (17.6.4.7).

17.6.4 Constraints on programs [constraints]

17.6.4.1 Overview [constraints.overview]

1 This section describes restrictions on C++ programs that use the facilities of the C++ standard library. The following subclauses specify constraints on the program's use of namespaces (17.6.4.2.1), its use of various reserved names (17.6.4.3), its use of headers (17.6.4.4), its use of standard library classes as base classes (17.6.4.5), its definitions of replacement functions (17.6.4.6), and its installation of handler functions during execution (17.6.4.7).

17.6.4.2 Namespace use [namespace.constraints]

17.6.4.2.1 Namespace `std` [namespace.std]

1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `std` or to a namespace within namespace `std` unless otherwise specified. A program may add a concept map for any standard library concept or a template specialization for any standard library template to namespace `std` only if the declaration depends on a user-defined type of external linkage and the specialization meets the standard library requirements for the original template and is not explicitly prohibited.[174]

2 The behavior of a C++ program is undefined if it declares

- an explicit specialization of any member function of a standard library class template, or
- an explicit specialization of any member function template of a standard library class or class template, or
- an explicit or partial specialization of any member class template of a standard library class or class template.

172) This is the same as the Standard C library.
173) The only reliable way to declare an object or function signature from the Standard C library is by including the header that declares it, notwithstanding the latitude granted in 7.1.7 of the C Standard.
174) Any library code that instantiates other library templates must be prepared to work adequately with any user-supplied specialization that meets the minimum requirements of the Standard.

A program may explicitly instantiate a template defined in the standard library only if the declaration depends on the name of a user-defined type of external linkage and the instantiation meets the standard library requirements for the original template.

3 A translation unit shall not declare namespace `std` to be an inline namespace (7.3.1).

17.6.4.2.2 Namespace `posix` [namespace.posix]

1 The behavior of a C++ program is undefined if it adds declarations or definitions to namespace `posix` or to a namespace within namespace `posix` unless otherwise specified. The namespace `posix` is reserved for use by ISO/IEC 9945 and other POSIX standards.

17.6.4.3 Reserved names [reserved.names]

17.6.4.3.1 General [reserved.names.general]

1 The C++ standard library reserves the following kinds of names:

- macros
- global names
- names with external linkage

2 If a program declares or defines a name in a context where it is reserved, other than as explicitly allowed by this Clause, its behavior is undefined.

17.6.4.3.2 Macro names [macro.names]

1 A translation unit that includes a standard library header shall not `#define` or `#undef` names declared in any standard library header.

2 A translation unit shall not `#define` or `#undef` names lexically identical to keywords.

17.6.4.3.3 Global names [global.names]

1 Certain sets of names and function signatures are always reserved to the implementation:

- Each name that contains a double underscore `__` or begins with an underscore followed by an uppercase letter (2.11) is reserved to the implementation for any use.
- Each name that begins with an underscore is reserved to the implementation for use as a name in the global namespace.[175]

17.6.4.3.4 External linkage [extern.names]

1 Each name declared as an object with external linkage in a header is reserved to the implementation to designate that library object with external linkage,[176] both in namespace `std` and in the global namespace.

2 Each global function signature declared with external linkage in a header is reserved to the implementation to designate that function signature with external linkage.[177]

---

175) Such names are also reserved in namespace `::std` (17.6.4.3).
176) The list of such reserved names includes `errno`, declared or defined in `<cerrno>`.
177) The list of such reserved function signatures with external linkage includes `setjmp(jmp_buf)`, declared or defined in `<csetjmp>`, and `va_end(va_list)`, declared or defined in `<cstdarg>`.

3 Each name having two consecutive underscores (2.11) is reserved to the implementation for use as a name with both `extern "C"` and `extern "C++"` linkage.

4 Each name from the Standard C library declared with external linkage is reserved to the implementation for use as a name with `extern "C"` linkage, both in namespace std and in the global namespace.

5 Each function signature from the Standard C library declared with external linkage is reserved to the implementation for use as a function signature with both `extern "C"` and `extern "C++"` linkage,[178] or as a name of namespace scope in the global namespace.

17.6.4.3.5 Types [extern.types]

1 For each type T from the Standard C library,[179] the types `::T` and `std::T` are reserved to the implementation and, when defined, `::T` shall be identical to `std::T`.

17.6.4.3.6 User-defined literal suffixes [usrlit.suffix]

1 Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

17.6.4.4 Headers [alt.headers]

1 If a file with a name equivalent to the derived file name for one of the C++ standard library headers is not provided as part of the implementation, and a file with that name is placed in any of the standard places for a source file to be included (16.2), the behavior is undefined.

17.6.4.5 Derived classes [derived.classes]

1 Virtual member function signatures defined for a base class in the C++ Standard library may be overridden in a derived class defined in the program (10.3).

17.6.4.6 Replacement functions [replacement.functions]

1 Clauses 18 through 30 and Annex D describe the behavior of numerous functions defined by the C++ standard library. Under some circumstances, however, certain of these function descriptions also apply to replacement functions defined in the program (17.3).

2 A C++ program may provide the definition for any of eight dynamic memory allocation function signatures declared in header `<new>` (3.7.4, Clause 18):

- `operator new(std::size_t)`
- `operator new(std::size_t, const std::nothrow_t&)`
- `operator new[](std::size_t)`
- `operator new[](std::size_t, const std::nothrow_t&)`
- `operator delete(void*)`
- `operator delete(void*, const std::nothrow_t&)`
- `operator delete[](void*)`
- `operator delete[](void*, const std::nothrow_t&)`

178) The function signatures declared in `<cuchar>`, `<cwchar>`, and `<cwctype>` are always reserved, notwithstanding the restrictions imposed in subclause 4.5.1 of Amendment 1 to the C Standard for these headers.

179) These types are `clock_t`, `div_t`, `FILE`, `fpos_t`, `lconv`, `ldiv_t`, `mbstate_t`, `ptrdiff_t`, `sig_atomic_t`, `size_t`, `time_t`, `tm`, `va_list`, `wctrans_t`, `wctype_t`, and `wint_t`.

3 The program's definitions are used instead of the default versions supplied by the implementation (18.5). Such replacement occurs prior to program startup (3.2, 3.6). The program's definitions shall not be specified as `inline`. No diagnostic is required.

17.6.4.7 Handler functions [handler.functions]

1 The C++ standard library provides default versions of the following handler functions (Clause 18):

— `unexpected_handler`

— `terminate_handler`

2 A C++ program may install different handler functions during execution, by supplying a pointer to a function defined in the program or the library as an argument to (respectively):

— `set_new_handler`

— `set_unexpected`

— `set_terminate`

See also: subclauses 18.5.2, Storage allocation errors, and 18.7, Exception handling.

17.6.4.8 Other functions [res.on.functions]

1 In certain cases (replacement functions, handler functions, operations on types used to instantiate standard library template components), the C++ standard library depends on components supplied by a C++ program. If these components do not meet their requirements, the Standard places no requirements on the implementation.

2 In particular, the effects are undefined in the following cases:

— for replacement functions (18.5.1), if the installed replacement function does not implement the semantics of the applicable *Required behavior:* paragraph.

— for handler functions (18.5.2.2, 18.7.3.1, 18.7.2.2), if the installed handler function does not implement the semantics of the applicable *Required behavior:* paragraph

— for types used as template arguments when instantiating a template component, if the operations on the type do not implement the semantics of the applicable **Requirements** subclause (20.7.2.2, 23.1.6, 24.1, 26.1). Operations on such types can report a failure by throwing an exception unless otherwise specified.

— if any replacement function or handler function or destructor operation throws an exception, unless specifically allowed in the applicable *Required behavior:* paragraph.

— if an incomplete type (3.9) is used as a template argument when instantiating a template component, unless specifically allowed for that component.

17.6.4.9 Function arguments [res.on.arguments]

1 Each of the following statements applies to all arguments to functions defined in the C++ standard library, unless explicitly stated otherwise.

— If an argument to a function has an invalid value (such as a value outside the domain of the function, or a pointer invalid for its intended use), the behavior is undefined.

— If a function argument is described as being an array, the pointer actually passed to the function shall have a value such that all address computations and accesses to objects (that would be valid if the pointer did point to the first element of such an array) are in fact valid.

17.6.4.10 Shared objects and the library [res.on.objects]

1 The behavior of a program is undefined if calls to standard library functions from different threads may introduce a data race. The conditions under which this may occur are specified in 17.6.5.7.

17.6.4.11 Required paragraph [res.on.required]

1 Violation of the preconditions specified in a function's *Required behavior:* paragraph results in undefined behavior unless the function's *Throws:* paragraph specifies throwing an exception when the precondition is violated.

17.6.5 Conforming implementations [conforming]

17.6.5.1 Overview [conforming.overview]

1 This section describes the constraints upon, and latitude of, implementations of the C++ standard library.

2 An implementation's use of headers is discussed in 17.6.5.2, its use of macros in 17.6.5.3, global functions in 17.6.5.4, member functions in 17.6.5.5, data race avoidance in 17.6.5.7, access specifiers in 17.6.5.8, class derivation in 17.6.5.9, and exceptions in 17.6.5.10.

17.6.5.2 Headers [res.on.headers]

1 A C++ header may include other C++ headers.[180]

2 Certain types and macros are defined in more than one header. Every such entity shall be defined such that any header that defines it may be included after any other header that also defines it (3.2).

3 The C standard headers (D.5) shall include only their corresponding C++ standard header, as described in 17.6.2.3.

17.6.5.3 Restrictions on macro definitions [res.on.macro.definitions]

1 The names and global function signatures described in 17.6.2.2 are reserved to the implementation.

2 All object-like macros defined by the C standard library and described in this Clause as expanding to integral constant expressions are also suitable for use in `#if` preprocessing directives, unless explicitly stated otherwise.

17.6.5.4 Global and non-member functions [global.functions]

1 It is unspecified whether any global or non-member functions in the C++ standard library are defined as `inline` (7.1.2).

2 A call to a global or non-member function signature described in Clauses 18 through 30 and Annex D shall behave as if the implementation declared no additional global or non-member function signatures.[181]

3 An implementation shall not declare a global or non-member function signature with additional default arguments.

180) C++ headers must include a C++ header that contains any needed definition (3.2).
181) A valid C++ program always calls the expected library global or non-member function. An implementation may also define additional global or non-member functions that would otherwise not be called by a valid C++ program.

4 Unless otherwise specified, global and non-member functions in the standard library shall not use functions from another namespace which are found through *argument-dependent name lookup* (3.4.2). [*Note:* The phrase "unless otherwise specified" is intended to allow argument-dependent lookup in cases like that of `ostream_iterator`s: *Effects:*

```
*out_stream << value;
if (delim != 0)
  *out_stream << delim;
return (*this);
```

— *end note*]

17.6.5.5 Member functions [member.functions]

1 It is unspecified whether any member functions in the C++ standard library are defined as `inline` (7.1.2).

2 An implementation may declare additional non-virtual member function signatures within a class:

— by adding arguments with default values to a member function signature;[182] [*Note:* An implementation may not add arguments with default values to virtual, global, or non-member functions. — *end note*]

— by replacing a member function signature with default values by two or more member function signatures with equivalent behavior; and

— by adding a member function signature for a member function name.

3 A call to a member function signature described in the C++ standard library behaves as if the implementation declares no additional member function signatures.[183]

17.6.5.6 Reentrancy [reentrancy]

1 Except where explicitly specified in this standard, it is implementation defined which functions in the C++ standard library may be recursively reentered.

17.6.5.7 Data race avoidance [res.on.data.races]

1 This section specifies requirements that implementations shall meet to prevent data races (1.10). Every standard library function shall meet each requirement unless otherwise specified. Implementations may prevent data races in cases other than those specified below.

2 A C++ standard library function shall not directly or indirectly access objects (1.10) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including `this`.

3 A C++ standard library function shall not directly or indirectly modify objects (1.10) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments, including `this`.

4 [*Note:* This means, for example, that implementations can't use a static object for internal purposes without synchronization because it could cause a data race even in programs that do not explicitly share objects between threads. — *end note*]

5 A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

182) Hence, the address of a member function of a class in the C++ standard library has an unspecified type.
183) A valid C++ program always calls the expected library member function, or one with equivalent behavior. An implementation may also define additional member functions that would otherwise not be called by a valid C++ program.

6 Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

7 Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible (1.10) to users.

8 [*Note:* This allows implementations to parallelize operations if there are no visible side effects. — *end note*]

17.6.5.8 Protection within classes [protection.within.classes]

1 It is unspecified whether any function signature or class described in Clauses 18 through 30 and Annex D is a `friend` of another class in the C++ standard library.

17.6.5.9 Derived classes [derivation]

1 An implementation may derive any class in the C++ standard library from a class with a name reserved to the implementation.

2 Certain classes defined in the C++ standard library are required to be derived from other classes in the C++ standard library. An implementation may derive such a class directly from the required base or indirectly through a hierarchy of base classes with names reserved to the implementation.

3 In any case:

— Every base class described as `virtual` shall be virtual;

— Every base class described as non-`virtual` shall not be virtual;

— Unless explicitly stated otherwise, types with distinct names shall be distinct types.[184]

17.6.5.10 Restrictions on exception handling [res.on.exception.handling]

1 Any of the functions defined in the C++ standard library can report a failure by throwing an exception of a type described in its **Throws:** paragraph or its *exception-specification* (15.4). An implementation may strengthen the *exception-specification* for a non-virtual function by removing listed exceptions.[185]

2 A function may throw an object of a type not listed in its **Throws** clause if its type is derived from a type named in the **Throws** clause and would be caught by an exception handler for the base type.

3 Functions from the C standard library shall not throw exceptions[186] except when such a function calls a program-supplied function that throws an exception.[187]

4 Destructor operations defined in the C++ standard library shall not throw exceptions. Any other functions defined in the C++ standard library that do not have an *exception-specification* may throw implementation-

184) There is an implicit exception to this rule for types that are described as synonyms for basic integral types, such as `size_t` (18.1) and `streamoff` (27.4.1).

185) That is, an implementation of the function will have an explicit *exception-specification* that lists fewer exceptions than those specified in this International Standard. It may not, however, change the types of exceptions listed in the *exception-specification* from those specified, nor add others.

186) That is, the C library functions all have a `throw()` *exception-specification*. This allows implementations to make performance optimizations based on the absence of exceptions at runtime.

187) The functions `qsort()` and `bsearch()` (25.4) meet this condition.

defined exceptions unless otherwise specified.[188] An implementation may strengthen this implicit *exception-specification* by adding an explicit one.[189]

17.6.5.11 Restrictions on storage of pointers [res.on.pointer.storage]

1 Objects constructed by the standard library that may hold a user-supplied pointer value or an integer of type `std::intptr_t` shall store such values in a traceable pointer location (3.7.4.3). [*Note:* Other libraries are strongly encouraged to do the same, since not doing so may result in accidental use of pointers that are not safely derived. Libraries that store pointers outside the user's address space should make it appear that they are stored and retrieved from a traceable pointer location. — *end note*]

17.6.5.12 Value of error codes [value.error.codes]

1 Certain functions in the C++ standard library report errors via a `std::error_code` (19.4.2.2) object. That object's `category()` member shall return a reference to `std::system_category` for errors originating from the operating system, or a reference to an implementation-defined `error_category` object for errors originating elsewhere. The implementation shall define the possible values of `value()` for each of these error categories. [*Example:* For operating systems that are based on POSIX, implementations are encouraged to define the `std::system_category` values as identical to the POSIX `errno` values, with additional values as defined by the operating system's documentation. Implementations for operating systems that are not based on POSIX are encouraged to define values identical to the operating system's values. For errors that do not originate from the operating system, the implementation may provide enums for the associated values. — *end example*]

188) In particular, they can report a failure to allocate storage by throwing an exception of type `bad_alloc`, or a class derived from `bad_alloc` (18.5.2.1). Library implementations are encouraged (but not required) to report errors by throwing exceptions from (or derived from) the standard exception classes (18.5.2.1, 18.7, 19.1).

189) That is, an implementation may provide an explicit *exception-specification* that defines the subset of "any" exceptions thrown by that function. This implies that the implementation may list implementation-defined types in such an *exception-specification*.

18 Language support library [language.support]

1 This Clause describes the function signatures that are called implicitly, and the types of objects generated implicitly, during the execution of some C++ programs. It also describes the headers that declare these function signatures and define any related types.

2 The following subclauses describe common type definitions used throughout the library, characteristics of the predefined types, functions supporting start and termination of a C++ program, support for dynamic memory management, support for dynamic type identification, support for exception processing, support for initializer lists, and other runtime support, as summarized in Table 16.

Table 16 — Language support library summary

Subclause		Header(s)
18.1	Types	`<cstddef>`

18.2	Implementation properties	`<limits>`
		`<climits>`
		`<cfloat>`

18.3	Integer types	`<cstdint>`
18.4	Start and termination	`<cstdlib>`
18.5	Dynamic memory management	`<new>`
18.6	Type identification	`<typeinfo>`
18.8	Initializer lists	`<initializer_list>`
18.7	Exception handling	`<exception>`

18.9	Other runtime support	`<cstdarg>`
		`<csetjmp>`
		`<ctime>`
		`<csignal>`
		`<cstdlib>`
		`<cstdbool>`

18.1 Types [support.types]

1 Table 17) describes the header `<cstddef>`.

Table 17 — Header `<cstddef>` synopsis

Type	Name(s)	
Macros:	`NULL`	`offsetof`
Types:	`ptrdiff_t`	`size_t`
	`max_align_t`	`nullptr_t`

2 The contents are the same as the Standard C library header `<stddef.h>`, with the following changes:

3 The macro `NULL` is an implementation-defined C++ null pointer constant in this International Standard (4.10).[190]

4 The macro `offsetof`(*type*, *member-designator*) accepts a restricted set of *type* arguments in this International Standard. If *type* is not a standard-layout class (Clause 9), the results are undefined.[191] The expression `offsetof`(*type*, *member-designator*) is never type-dependent (14.6.2.2) and it is value-dependent (14.6.2.3) if and only if *type* is dependent. The result of applying the `offsetof` macro to a field that is a static data member or a function member is undefined.

5 The type `max_align_t` is a POD type whose alignment requirement is at least as great as that of every scalar type, and whose alignment requirement is supported in every context.

6 `nullptr_t` is defined as follows:

```
namespace std {
  typedef decltype(nullptr) nullptr_t;
}
```

The type for which `nullptr_t` is a synonym has the characteristics described in 3.9.1 and 4.10. [*Note:* Although `nullptr`'s address cannot be taken, the address of another `nullptr_t` object that is an lvalue can be taken. — *end note*]

See also: subclause 5.3.3, Sizeof, subclause 5.7, Additive operators, subclause 12.5, Free store, and ISO C 7.1.6.

18.2 Implementation properties [support.limits]

1 The headers `<limits>`, `<climits>`, `<cfloat>`, and `<cinttypes>` supply characteristics of implementation-dependent arithmetic types (3.9.1).

18.2.1 Numeric limits [limits]

1 The `numeric_limits` component provides a C++ program with information about various properties of the implementation's representation of the arithmetic types.

2 Specializations shall be provided for each arithmetic type, both floating point and integer, including `bool`. The member `is_specialized` shall be `true` for all such specializations of `numeric_limits`.

3 For all members declared `static constexpr` in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as constant expressions.

4 Non-arithmetic standard types, such as `complex<T>` (26.3.2), shall not have specializations.

Header `<limits>` synopsis

```
namespace std {
  template<Regular T> class numeric_limits;
  enum float_round_style;
  enum float_denorm_style;

  template<> class numeric_limits<bool>;

  template<> class numeric_limits<char>;
  template<> class numeric_limits<signed char>;
  template<> class numeric_limits<unsigned char>;
  template<> class numeric_limits<char16_t>;
```

190) Possible definitions include `0` and `0L`, but not `(void*)0`.
191) Note that `offsetof` is required to work as specified even if unary `operator&` is overloaded for any of the types involved.

```
  template<> class numeric_limits<char32_t>;
  template<> class numeric_limits<wchar_t>;

  template<> class numeric_limits<short>;
  template<> class numeric_limits<int>;
  template<> class numeric_limits<long>;
  template<> class numeric_limits<long long>;
  template<> class numeric_limits<unsigned short>;
  template<> class numeric_limits<unsigned int>;
  template<> class numeric_limits<unsigned long>;
  template<> class numeric_limits<unsigned long long>;

  template<> class numeric_limits<float>;
  template<> class numeric_limits<double>;
  template<> class numeric_limits<long double>;
}
```

18.2.1.1 Class template `numeric_limits` [numeric.limits]

```
namespace std {
  template<Regular T> class numeric_limits {
  public:
    static constexpr bool is_specialized = false;
    static constexpr T min() throw() { return T(); }
    static constexpr T max() throw() { return T(); }
    static constexpr T lowest() throw() { return T(); }

    static constexpr int  digits = 0;
    static constexpr int  digits10 = 0;
    static constexpr int  max_digits10 = 0;
    static constexpr bool is_signed = false;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;
    static constexpr int  radix = 0;
    static constexpr T epsilon() throw() { return T(); }
    static constexpr T round_error() throw() { return T(); }

    static constexpr int  min_exponent = 0;
    static constexpr int  min_exponent10 = 0;
    static constexpr int  max_exponent = 0;
    static constexpr int  max_exponent10 = 0;

    static constexpr bool has_infinity = false;
    static constexpr bool has_quiet_NaN = false;
    static constexpr bool has_signaling_NaN = false;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss = false;
    static constexpr T infinity() throw() { return T(); }
    static constexpr T quiet_NaN() throw() { return T(); }
    static constexpr T signaling_NaN() throw() { return T(); }
    static constexpr T denorm_min() throw() { return T(); }

    static constexpr bool is_iec559 = false;
    static constexpr bool is_bounded = false;
    static constexpr bool is_modulo = false;
```

```
    static constexpr bool traps = false;
    static constexpr bool tinyness_before = false;
    static constexpr float_round_style round_style = round_toward_zero;
  };

  template<class T> class numeric_limits<const T>;
  template<class T> class numeric_limits<volatile T>;
  template<class T> class numeric_limits<const volatile T>;
}
```

1 The default `numeric_limits<T>` template shall have all members, but with 0 or `false` values.

2 The value of each member of a specialization of `numeric_limits` on a *cv*-qualified type `cv T` shall be equal to the value of the corresponding member of the specialization on the unqualified type `T`.

18.2.1.2 `numeric_limits` members [numeric.limits.members]

`static constexpr T min() throw();`

1 Minimum finite value.[192]

2 For floating types with denormalization, returns the minimum positive normalized value.

3 Meaningful for all specializations in which `is_bounded != false`, or `is_bounded == false && is_signed == false`.

`static constexpr T max() throw();`

4 Maximum finite value.[193]

5 Meaningful for all specializations in which `is_bounded != false`.

`static constexpr T lowest() throw();`

6 A finite value `x` such that there is no other finite value `y` where `y < x`.[194]

7 Meaningful for all specializations in which `is_bounded != false`.

`static constexpr int  digits;`

8 Number of `radix` digits that can be represented without change.

9 For integer types, the number of non-sign bits in the representation.

10 For floating point types, the number of `radix` digits in the mantissa.[195]

`static constexpr int  digits10;`

11 Number of base 10 digits that can be represented without change.[196]

12 Meaningful for all specializations in which `is_bounded != false`.

`static constexpr int max_digits10;`

192) Equivalent to `CHAR_MIN`, `SHRT_MIN`, `FLT_MIN`, `DBL_MIN`, etc.
193) Equivalent to `CHAR_MAX`, `SHRT_MAX`, `FLT_MAX`, `DBL_MAX`, etc.
194) `lowest()` is necessary because not all floating-point representations have a smallest (most negative) value that is the negative of the largest (most positive) finite value.
195) Equivalent to `FLT_MANT_DIG`, `DBL_MANT_DIG`, `LDBL_MANT_DIG`.
196) Equivalent to `FLT_DIG`, `DBL_DIG`, `LDBL_DIG`.

13 Number of base 10 digits required to ensure that values which differ are always differentiated.

14 Meaningful for all floating point types.

```
static constexpr bool is_signed;
```

15 True if the type is signed.

16 Meaningful for all specializations.

```
static constexpr bool is_integer;
```

17 True if the type is integer.

18 Meaningful for all specializations.

```
static constexpr bool is_exact;
```

19 True if the type uses an exact representation. All integer types are exact, but not all exact types are integer. For example, rational and fixed-exponent representations are exact but not integer.

20 Meaningful for all specializations.

```
static constexpr int  radix;
```

21 For floating types, specifies the base or radix of the exponent representation (often 2).[197]

22 For integer types, specifies the base of the representation.[198]

23 Meaningful for all specializations.

```
static constexpr T epsilon() throw();
```

24 Machine epsilon: the difference between 1 and the least value greater than 1 that is representable.[199]

25 Meaningful for all floating point types.

```
static constexpr T round_error() throw();
```

26 Measure of the maximum rounding error.[200]

```
static constexpr int  min_exponent;
```

27 Minimum negative integer such that `radix` raised to the power of one less than that integer is a normalized floating point number.[201]

28 Meaningful for all floating point types.

```
static constexpr int  min_exponent10;
```

29 Minimum negative integer such that 10 raised to that power is in the range of normalized floating point numbers.[202]

30 Meaningful for all floating point types.

```
static constexpr int  max_exponent;
```

197) Equivalent to `FLT_RADIX`.
198) Distinguishes types with bases other than 2 (e.g. BCD).
199) Equivalent to `FLT_EPSILON`, `DBL_EPSILON`, `LDBL_EPSILON`.
200) Rounding error is described in ISO/IEC 10967-1 Language independent arithmetic - Part 1 Section 5.2.8 and Annex A Rationale Section A.5.2.8 - Rounding constants.
201) Equivalent to `FLT_MIN_EXP`, `DBL_MIN_EXP`, `LDBL_MIN_EXP`.
202) Equivalent to `FLT_MIN_10_EXP`, `DBL_MIN_10_EXP`, `LDBL_MIN_10_EXP`.

31 Maximum positive integer such that `radix` raised to the power one less than that integer is a representable finite floating point number.[203]

32 Meaningful for all floating point types.

```
static constexpr int  max_exponent10;
```

33 Maximum positive integer such that 10 raised to that power is in the range of representable finite floating point numbers.[204]

34 Meaningful for all floating point types.

```
static constexpr bool has_infinity;
```

35 True if the type has a representation for positive infinity.

36 Meaningful for all floating point types.

37 Shall be `true` for all specializations in which `is_iec559 != false`.

```
static constexpr bool has_quiet_NaN;
```

38 True if the type has a representation for a quiet (non-signaling) "Not a Number."[205]

39 Meaningful for all floating point types.

40 Shall be `true` for all specializations in which `is_iec559 != false`.

```
static constexpr bool has_signaling_NaN;
```

41 True if the type has a representation for a signaling "Not a Number."[206]

42 Meaningful for all floating point types.

43 Shall be `true` for all specializations in which `is_iec559 != false`.

```
static constexpr float_denorm_style has_denorm;
```

44 `denorm_present` if the type allows denormalized values (variable number of exponent bits)[207], `denorm_absent` if the type does not allow denormalized values, and `denorm_indeterminate` if it is indeterminate at compile time whether the type allows denormalized values.

45 Meaningful for all floating point types.

```
static constexpr bool has_denorm_loss;
```

46 True if loss of accuracy is detected as a denormalization loss, rather than as an inexact result.[208]

```
static constexpr T infinity() throw();
```

47 Representation of positive infinity, if available.[209]

48 Meaningful for all specializations for which `has_infinity != false`. Required in specializations for which `is_iec559 != false`.

```
static constexpr T quiet_NaN() throw();
```

203) Equivalent to `FLT_MAX_EXP`, `DBL_MAX_EXP`, `LDBL_MAX_EXP`.
204) Equivalent to `FLT_MAX_10_EXP`, `DBL_MAX_10_EXP`, `LDBL_MAX_10_EXP`.
205) Required by LIA-1.
206) Required by LIA-1.
207) Required by LIA-1.
208) See IEC 559.
209) Required by LIA-1.

49 Representation of a quiet "Not a Number," if available.[210]

50 Meaningful for all specializations for which `has_quiet_NaN != false`. Required in specializations for which `is_iec559 != false`.

```
static constexpr T signaling_NaN() throw();
```

51 Representation of a signaling "Not a Number," if available.[211]

52 Meaningful for all specializations for which `has_signaling_NaN != false`. Required in specializations for which `is_iec559 != false`.

```
static constexpr T denorm_min() throw();
```

53 Minimum positive denormalized value.[212]

54 Meaningful for all floating point types.

55 In specializations for which `has_denorm == false`, returns the minimum positive normalized value.

```
static constexpr bool is_iec559;
```

56 True if and only if the type adheres to IEC 559 standard.[213]

57 Meaningful for all floating point types.

```
static constexpr bool is_bounded;
```

58 True if the set of values representable by the type is finite.[214] [*Note:* All built-in types are bounded. This member would be false for arbitrary precision types. — *end note*]

59 Meaningful for all specializations.

```
static constexpr bool is_modulo;
```

60 True if the type is modulo.[215] A type is modulo if, for any operation involving `+`, `-`, or `*` on values of that type whose result would fall outside the range `[min(),max()]`, the value returned differs from the true value by an integer multiple of `max() - min() + 1`.

61 On most machines, this is `false` for floating types, `true` for unsigned integers, and `true` for signed integers.

62 Meaningful for all specializations.

```
static constexpr bool traps;
```

63 `true` if, at program startup, there exists a value of the type that would cause an arithmetic operation using that value to trap.[216]

64 Meaningful for all specializations.

```
static constexpr bool tinyness_before;
```

210) Required by LIA-1.
211) Required by LIA-1.
212) Required by LIA-1.
213) International Electrotechnical Commission standard 559 is the same as IEEE 754.
214) Required by LIA-1.
215) Required by LIA-1.
216) Required by LIA-1.

65 `true` if tinyness is detected before rounding.[217]

66 Meaningful for all floating point types.

```
static constexpr float_round_style round_style;
```

67 The rounding style for the type.[218]

68 Meaningful for all floating point types. Specializations for integer types shall return `round_toward_zero`.

18.2.1.3 Type `float_round_style` [round.style]

```
namespace std {
  enum float_round_style {
    round_indeterminate       = -1,
    round_toward_zero         =  0,
    round_to_nearest          =  1,
    round_toward_infinity     =  2,
    round_toward_neg_infinity =  3
  };
}
```

1 The rounding mode for floating point arithmetic is characterized by the values:

— `round_indeterminate` if the rounding style is indeterminable

— `round_toward_zero` if the rounding style is toward zero

— `round_to_nearest` if the rounding style is to the nearest representable value

— `round_toward_infinity` if the rounding style is toward infinity

— `round_toward_neg_infinity` if the rounding style is toward negative infinity

18.2.1.4 Type `float_denorm_style` [denorm.style]

```
namespace std {
  enum float_denorm_style {
    denorm_indeterminate = -1,
    denorm_absent = 0,
    denorm_present = 1
  };
}
```

1 The presence or absence of denormalization (variable number of exponent bits) is characterized by the values:

— `denorm_indeterminate` if it cannot be determined whether or not the type allows denormalized values

— `denorm_absent` if the type does not allow denormalized values

— `denorm_present` if the type does allow denormalized values

217) Refer to IEC 559. Required by LIA-1.

218) Equivalent to `FLT_ROUNDS`. Required by LIA-1.

18.2.1.5 `numeric_limits` specializations [numeric.special]

1 All members shall be provided for all specializations. However, many values are only required to be meaningful under certain conditions (for example, `epsilon()` is only meaningful if `is_integer` is `false`). Any value that is not "meaningful" shall be set to 0 or `false`.

2 [*Example:*

```
namespace std {
  template<> class numeric_limits<float> {
  public:
    static constexpr bool is_specialized = true;

    inline static constexpr float min() throw() { return 1.17549435E-38F; }
    inline static constexpr float max() throw() { return 3.40282347E+38F; }
    inline static constexpr float lowest() throw() { return -3.40282347E+38F; }

    static constexpr int digits   = 24;
    static constexpr int digits10 =  6;
    static constexpr int max_digits10 =  9;

    static constexpr bool is_signed  = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact   = false;

    static constexpr int radix = 2;
    inline static constexpr float epsilon() throw()     { return 1.19209290E-07F; }
    inline static constexpr float round_error() throw() { return 0.5F; }

    static constexpr int min_exponent   = -125;
    static constexpr int min_exponent10 = - 37;
    static constexpr int max_exponent   = +128;
    static constexpr int max_exponent10 = + 38;

    static constexpr bool has_infinity             = true;
    static constexpr bool has_quiet_NaN            = true;
    static constexpr bool has_signaling_NaN        = true;
    static constexpr float_denorm_style has_denorm = denorm_absent;
    static constexpr bool has_denorm_loss          = false;

    inline static constexpr float infinity()      throw() { return ...; }
    inline static constexpr float quiet_NaN()     throw() { return ...; }
    inline static constexpr float signaling_NaN() throw() { return ...; }
    inline static constexpr float denorm_min()    throw() { return min(); }

    static constexpr bool is_iec559  = true;
    static constexpr bool is_bounded = true;
    static constexpr bool is_modulo  = false;
    static constexpr bool traps      = true;
    static constexpr bool tinyness_before = true;

    static constexpr float_round_style round_style = round_to_nearest;
  };
}
```

— *end example*]

3 The specialization for `bool` shall be provided as follows:

```
namespace std {
   template<> class numeric_limits<bool> {
   public:
     static constexpr bool is_specialized = true;
     static constexpr bool min() throw() { return false; }
     static constexpr bool max() throw() { return true; }
     static constexpr bool lowest() throw() { return false; }

     static constexpr int  digits = 1;
     static constexpr int  digits10 = 0;
     static constexpr int  max_digits10 = 0;

     static constexpr bool is_signed = false;
     static constexpr bool is_integer = true;
     static constexpr bool is_exact = true;
     static constexpr int  radix = 2;
     static constexpr bool epsilon() throw() { return 0; }
     static constexpr bool round_error() throw() { return 0; }

     static constexpr int  min_exponent = 0;
     static constexpr int  min_exponent10 = 0;
     static constexpr int  max_exponent = 0;
     static constexpr int  max_exponent10 = 0;

     static constexpr bool has_infinity = false;
     static constexpr bool has_quiet_NaN = false;
     static constexpr bool has_signaling_NaN = false;
     static constexpr float_denorm_style has_denorm = denorm_absent;
     static constexpr bool has_denorm_loss = false;
     static constexpr bool infinity() throw() { return 0; }
     static constexpr bool quiet_NaN() throw() { return 0; }
     static constexpr bool signaling_NaN() throw() { return 0; }
     static constexpr bool denorm_min() throw() { return 0; }

     static constexpr bool is_iec559 = false;
     static constexpr bool is_bounded = true;
     static constexpr bool is_modulo = false;

     static constexpr bool traps = false;
     static constexpr bool tinyness_before = false;
     static constexpr float_round_style round_style = round_toward_zero;
   };
}
```

18.2.2 C Library [c.limits]

1 Table 18 describes the header `<climits>`.

2 The contents are the same as the Standard C library header `<limits.h>`. [*Note:* The types of the constants defined by macros in `<climits>` are not required to match the types to which the macros refer. — *end note*]

3 Table 19 describes the header `<cfloat>`.

4 The contents are the same as the Standard C library header `<float.h>`.

Table 18 — Header <climits> synopsis

Type	Name(s)				
Values:					
CHAR_BIT	INT_MAX	LONG_MAX	SCHAR_MIN	SHRT_MIN	ULLONG_MAX
CHAR_MAX	LLONG_MAX	LONG_MIN	SCHAR_MAX	UCHAR_MAX	ULONG_MAX
CHAR_MIN	LLONG_MIN	MB_LEN_MAX	SHRT_MAX	UINT_MAX	USHRT_MAX
INT_MIN					

Table 19 — Header <cfloat> synopsis

Type	Name(s)		
Values:			
DBL_DIG	DBL_MIN_EXP	FLT_MAX_EXP	LDBL_MANT_DIG
DBL_EPSILON	DECIMAL_DIG	FLT_MIN	LDBL_MAX_10_EXP
DBL_MANT_DIG	FLT_DIG	FLT_MIN_10_EXP	LDBL_MAX_EXP
DBL_MAX	FLT_EPSILON	FLT_MIN_EXP	LDBL_MAX
DBL_MAX_10_EXP	FLT_EVAL_METHOD	FLT_RADIX	LDBL_MIN
DBL_MAX_EXP	FLT_MANT_DIG	FLT_ROUNDS	LDBL_MIN_10_EXP
DBL_MIN	FLT_MAX	LDBL_DIG	LDBL_MIN_EXP
DBL_MIN_10_EXP	FLT_MAX_10_EXP	LDBL_EPSILON	

See also: ISO C 7.1.5, 5.2.4.2.2, 5.2.4.2.1.

18.3 Integer types [cstdint]

18.3.1 Header <cstdint> synopsis [cstdint.syn]

```
namespace std {
  typedef signed integer type int8_t;   // optional
  typedef signed integer type int16_t;  // optional
  typedef signed integer type int32_t;  // optional
  typedef signed integer type int64_t;  // optional

  typedef signed integer type int_fast8_t;
  typedef signed integer type int_fast16_t;
  typedef signed integer type int_fast32_t;
  typedef signed integer type int_fast64_t;

  typedef signed integer type int_least8_t;
  typedef signed integer type int_least16_t;
  typedef signed integer type int_least32_t;
  typedef signed integer type int_least64_t;

  typedef signed integer type intmax_t;
  typedef signed integer type intptr_t;         // optional

  typedef unsigned integer type uint8_t;        // optional
  typedef unsigned integer type uint16_t;       // optional
  typedef unsigned integer type uint32_t;       // optional
  typedef unsigned integer type uint64_t;       // optional

  typedef unsigned integer type uint_fast8_t;
```

```
  typedef unsigned integer type uint_fast16_t;
  typedef unsigned integer type uint_fast32_t;
  typedef unsigned integer type uint_fast64_t;

  typedef unsigned integer type uint_least8_t;
  typedef unsigned integer type uint_least16_t;
  typedef unsigned integer type uint_least32_t;
  typedef unsigned integer type uint_least64_t;

  typedef unsigned integer type uintmax_t;
  typedef unsigned integer type uintptr_t;      // optional
} // namespace std
```

1 The header also defines numerous macros of the form:

```
INT_[FAST LEAST]{8 16 32 64}_MIN
[U]INT_[FAST LEAST]{8 16 32 64}_MAX
INT{MAX PTR}_MIN
[U]INT{MAX PTR}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX
```

plus function macros of the form:

```
[U]INT{8 16 32 64 MAX}_C
```

2 The header defines all functions, types, and macros the same as C99 7.18. [*Note:* The macros defined by `<stdint>` are provided unconditionally. In particular, the symbols `__STDC_LIMIT_MACROS` and `__STDC_CONSTANT_MACROS` (mentioned in C99 footnotes 219, 220, and 222) play no role in C++. — *end note*]

18.3.2 The header `<stdint.h>` [stdinth]

1 The header behaves as if it includes the header `<cstdint>`, and provides sufficient *using* declarations to declare in the global namespace all type names defined in the header `<cstdint>`.

18.4 Start and termination [support.start.term]

1 Table 20 describes some of the contents of the header `<cstdlib>`.

Table 20 — Header `<cstdlib>` synopsis

Type	Name(s)		
Macros:	`EXIT_FAILURE`	`EXIT_SUCCESS`	
Functions:	`abort`	`atexit`	`at_quick_exit`
	`exit`	`quick_exit`	

2 The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

`abort(void)`

3 The function `abort()` has additional behavior in this International Standard:

— The program is terminated without executing destructors for objects of automatic, thread, or static storage duration and without calling the functions passed to `atexit()` (3.6.3).

```
extern "C" int atexit(void (*f)(void))
extern "C++" int atexit(void (*f)(void))
```

4 *Effects:* The `atexit()` functions register the function pointed to by `f` to be called without arguments at normal program termination.

5 *Implementation limits:* The implementation shall support the registration of at least 32 functions.

6 *Returns:* The `atexit()` function returns zero if the registration succeeds, nozero if it fails.

`exit(int status)`

7 The function `exit()` has additional behavior in this International Standard:

— First, objects with thread storage duration and associated with the current thread are destroyed. Next, objects with static storage duration are destroyed and functions registered by calling `atexit` are called.[219] See 3.6.3 for the order of destructions and calls. (Automatic objects are not destroyed as a result of calling `exit()`.)[220]

If control leaves a registered function called by `exit` because the function does not provide a handler for a thrown exception, `terminate()` shall be called.

— Next, all open C streams (as mediated by the function signatures declared in `<cstdio>`) with unwritten buffered data are flushed, all open C streams are closed, and all files created by calling `tmpfile()` are removed.[221]

— Finally, control is returned to the host environment. If `status` is zero or `EXIT_SUCCESS`, an implementation-defined form of the status *successful termination* is returned. If `status` is `EXIT_FAILURE`, an implementation-defined form of the status *unsuccessful termination* is returned. Otherwise the status returned is implementation-defined.[222]

8 The function `exit()` never returns to its caller.

```
extern "C" int at_quick_exit(void (*f)(void));
extern "C++" int at_quick_exit(void (*f)(void));
```

9 *Effects:* The `at_quick_exit()` functions register the function pointed to by `f` to be called without arguments when `quick_exit` is called. The `at_quick_exit()` functions shall be thread safe. [*Note:* The `at_quick_exit` registrations are distinct from the `atexit` registrations, and applications may need to call both registration functions with the same argument. — *end note*]

10 *Implementation limits:* The implementation shall support the registration of at least 32 functions.

11 *Returns:* zero if the registration succeeds, non-zero if it fails.

`void quick_exit(int status)`

12 *Effects:* Functions registered by calls to `at_quick_exit` are called in the reverse order of their registration, except that a function shall be called after any previously registered functions that had already been called at the time it was registered. Objects shall not be destroyed as a result of calling `quick_exit`.

If control leaves a registered function called by `quick_exit` because the function does not provide a handler for a thrown exception, `terminate()` shall be called.

219) A function is called for every time it is registered.
220) Objects with automatic storage duration are all destroyed in a program whose function `main()` contains no automatic objects and executes the call to `exit()`. Control can be transferred directly to such a `main()` by throwing an exception that is caught in `main()`.
221) Any C streams associated with `cin`, `cout`, etc (27.3) are flushed and closed when static objects are destroyed in the previous phase. The function `tmpfile()` is declared in `<cstdio>`.
222) The macros `EXIT_FAILURE` and `EXIT_SUCCESS` are defined in `<cstdlib>`.

After calling registered functions, `quick_exit` shall call `_Exit(status)`. [*Note:* The standard file buffers are not flushed. See: ISO C 7.20.4.4. — *end note*]

13 The function `quick_exit()` never returns to its caller.

See also: 3.6, 3.6.3, ISO C 7.10.4.

18.5 Dynamic memory management [support.dynamic]

1 The header `<new>` defines several functions that manage the allocation of dynamic storage in a program. It also defines components for reporting storage management errors.

Header `<new>` synopsis

```
namespace std {
  class bad_alloc;
  struct nothrow_t {};
  extern const nothrow_t nothrow;
  typedef void (*new_handler)();
  new_handler set_new_handler(new_handler new_p) throw();
}

void* operator new(std::size_t size) throw(std::bad_alloc);
void* operator new(std::size_t size, const std::nothrow_t&) throw();
void  operator delete(void* ptr) throw();
void  operator delete(void* ptr, const std::nothrow_t&) throw();
void* operator new[](std::size_t size) throw(std::bad_alloc);
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
void  operator delete[](void* ptr) throw();
void  operator delete[](void* ptr, const std::nothrow_t&) throw();

void* operator new  (std::size_t size, void* ptr) throw();
void* operator new[](std::size_t size, void* ptr) throw();
void  operator delete  (void* ptr, void*) throw();
void  operator delete[](void* ptr, void*) throw();
```

See also: 1.7, 3.7.4, 5.3.4, 5.3.5, 12.5, 20.7.

18.5.1 Storage allocation and deallocation [new.delete]

1 Except where otherwise specified, the provisions of (3.7.4) apply to the library versions of `operator new` and `operator delete`.

18.5.1.1 Single-object forms [new.delete.single]

```
void* operator new(std::size_t size) throw(std::bad_alloc);
```

1 *Effects:* The *allocation function* (3.7.4.1) called by a *new-expression* (5.3.4) to allocate `size` bytes of storage suitably aligned to represent any object of that size.

2 *Replaceable:* a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

3 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.7.4), or else throw a `bad_alloc` exception. This requirement is binding on a replacement version of this function.

4 *Default behavior:*

— Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function `malloc` is unspecified.

— Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to `set_new_handler()` was a null pointer, throw `bad_alloc`.

— Otherwise, the function calls the current *new_handler* (18.5.2.2). If the called function returns, the loop repeats.

— The loop terminates when an attempt to allocate the requested storage is successful or when a called *new_handler* function does not return.

```
void* operator new(std::size_t size, const std::nothrow_t&) throw();
```

5 *Effects:* Same as above, except that it is called by a placement version of a *new-expression* when a C ++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

6 *Replaceable:* a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

7 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.7.4), or else return a null pointer. This nothrow version of `operator new` returns a pointer obtained as if acquired from the (possibly replaced) ordinary version. This requirement is binding on a replacement version of this function.

8 *Default behavior:* Calls `operator new(size)`. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

9 [*Example:*

```
T* p1 = new T;                  // throws bad_alloc if it fails
T* p2 = new(nothrow) T;         // returns 0 if it fails
```

— *end example*]

```
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

10 *Effects:* The *deallocation function* (3.7.4.2) called by a *delete-expression* to render the value of `ptr` invalid.

11 *Replaceable:* a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

12 *Requires: ptr* shall be a null pointer or its value shall be a value returned by an earlier call to the (possibly replaced) `operator new(std::size_t)` or `operator new(std::size_t,const std::nothrow_t&)` which has not been invalidated by an intervening call to `operator delete(void*)`.

13 *Default behavior:* If `ptr` is null, does nothing. Otherwise, reclaims the storage allocated by the earlier call to `operator new`.

14 *Remarks:* It is unspecified under what conditions part or all of such reclaimed storage will be allocated by subsequent calls to `operator new` or any of `calloc`, `malloc`, or `realloc`, declared in `<cstdlib>`.

```
void operator delete(void* ptr, const std::nothrow_t&) throw();
```

15 *Effects:* The *deallocation function* (3.7.4.2) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the *new-expression* throws an exception.

16 *Replaceable:* a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

17 *Default behavior:* calls `operator delete(ptr)`.

18.5.1.2 Array forms [new.delete.array]

```
void* operator new[](std::size_t size) throw(std::bad_alloc);
```

1 *Effects:* The *allocation function* (3.7.4.1) called by the array form of a *new-expression* (5.3.4) to allocate `size` bytes of storage suitably aligned to represent any array object of that size or smaller.[223]

2 *Replaceable:* a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

3 *Required behavior:* Same as for `operator new(std::size_t)`. This requirement is binding on a replacement version of this function.

4 *Default behavior:* Returns `operator new(size)`.

```
void* operator new[](std::size_t size, const std::nothrow_t&) throw();
```

5 *Effects:* Same as above, except that it is called by a placement version of a *new-expression* when a C ++ program prefers a null pointer result as an error indication, instead of a `bad_alloc` exception.

6 *Replaceable:* a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

7 *Required behavior:* Return a non-null pointer to suitably aligned storage (3.7.4), or return a null pointer. This requirement is binding on a replacement version of this function.

8 *Default behavior:* Calls `operator new[](size)`. If the call returns normally, returns the result of that call. Otherwise, returns a null pointer.

```
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

9 *Effects:* The *deallocation function* (3.7.4.2) called by the array form of a *delete-expression* to render the value of `ptr` invalid.

10 *Replaceable:* a C++ program can define a function with this function signature that displaces the default version defined by the C++ standard library.

11 *Requires:* *ptr* shall be a null pointer or its value shall be the value returned by an earlier call to `operator new[](std::size_t)` or `operator new[](std::size_t,const std::nothrow_t&)` which has not been invalidated by an intervening call to `operator delete[](void*)`.

12 *Default behavior:* Calls `operator delete(ptr)`.

```
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

13 *Effects:* The *deallocation function* (3.7.4.2) called by the implementation to render the value of `ptr` invalid when the constructor invoked from a nothrow placement version of the array *new-expression* throws an exception.

---

223) It is not the direct responsibility of `operator new[](std::size_t)` or `operator delete[](void*)` to note the repetition count or element size of the array. Those operations are performed elsewhere in the array `new` and `delete` expressions. The array `new` expression, may, however, increase the `size` argument to `operator new[](std::size_t)` to obtain space to store supplemental information.

Replaceable: a C++ program may define a function with this function signature that displaces the default version defined by the C++ standard library.

Default behavior: calls `operator delete[](ptr)`.

18.5.1.3 Placement forms [new.delete.placement]

These functions are reserved, a C++ program may not define functions that displace the versions in the Standard C++ library (17.6.4). The provisions of (3.7.4) do not apply to these reserved placement forms of `operator new` and `operator delete`.

```
void* operator new(std::size_t size, void* ptr) throw();
```

Returns: `ptr`.

Remarks: Intentionally performs no other action.

[*Example:* This can be useful for constructing an object at a known address:

```
void* place = operator new(sizeof(Something));
Something* p = new (place) Something();
```

— *end example*]

```
void* operator new[](std::size_t size, void* ptr) throw();
```

Returns: `ptr`.

Remarks: Intentionally performs no other action.

```
void operator delete(void* ptr, void*) throw();
```

Effects: Intentionally performs no action.

Remarks: Default function called when any part of the initialization in a placement new expression that invokes the library's non-array placement operator new terminates by throwing an exception (5.3.4).

```
void operator delete[](void* ptr, void*) throw();
```

Effects: Intentionally performs no action.

Remarks: Default function called when any part of the initialization in a placement new expression that invokes the library's array placement operator new terminates by throwing an exception (5.3.4).

18.5.1.4 Data races [new.delete.dataraces]

The library versions of `operator new` and `operator delete`, user replacement versions of global `operator new` and `operator delete`, and the C standard library functions `calloc`, `malloc`, `realloc`, and `free` shall not introduce data races (1.10) as a result of concurrent calls from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

18.5.2 Storage allocation errors [alloc.errors]

18.5.2.1 Class `bad_alloc` [bad.alloc]

```
namespace std {
  class bad_alloc : public exception {
  public:
```

```
    bad_alloc() throw();
    bad_alloc(const bad_alloc&) throw();
    bad_alloc& operator=(const bad_alloc&) throw();
    virtual const char* what() const throw();
  };
}
```

1 The class `bad_alloc` defines the type of objects thrown as exceptions by the implementation to report a failure to allocate storage.

```
bad_alloc() throw();
```

2 *Effects:* Constructs an object of class `bad_alloc`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_alloc(const bad_alloc&) throw();
bad_alloc& operator=(const bad_alloc&) throw();
```

4 *Effects:* Copies an object of class `bad_alloc`.

```
virtual const char* what() const throw();
```

5 *Returns:* An implementation-defined NTBS.

18.5.2.2 Type `new_handler` [new.handler]

```
typedef void (*new_handler)();
```

1 The type of a *handler function* to be called by `operator new()` or `operator new[]()` (18.5.1) when they cannot satisfy a request for additional storage.

2 *Required behavior:* A `new_handler` shall perform one of the following:

- make more storage available for allocation and then return;
- throw an exception of type `bad_alloc` or a class derived from `bad_alloc`;
- call either `abort()` or `exit()`;

18.5.2.3 `set_new_handler` [set.new.handler]

```
new_handler set_new_handler(new_handler new_p) throw();
```

1 *Effects:* Establishes the function designated by `new_p` as the current `new_handler`.

2 *Returns:* 0 on the first call, the previous `new_handler` on subsequent calls.

18.6 Type identification [support.rtti]

1 The header `<typeinfo>` defines a type associated with type information generated by the implementation. It also defines two types for reporting dynamic type identification errors.

Header `<typeinfo>` synopsis

```
namespace std {
  class type_info;
  class type_index;
  template <class T> struct hash;
```

```
  template<>
  struct hash<type_index> : public std::unary_function<type_index, size_t> {
    size_t operator()(type_index index) const;
  }
  class bad_cast;
  class bad_typeid;
}
```

See also: 5.2.7, 5.2.8.

18.6.1 Class `type_info` [type.info]

```
namespace std {
  class type_info {
  public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    size_t hash_code() const throw();
    const char* name() const;

    type_info(const type_info& rhs) = delete;            // cannot be copied
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
  };
}
```

1 The class `type_info` describes type information generated by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoding rule, and collating sequence for types are all unspecified and may differ between programs.

```
bool operator==(const type_info& rhs) const;
```

2 *Effects:* Compares the current object with `rhs`.

3 *Returns:* `true` if the two values describe the same type.

```
bool operator!=(const type_info& rhs) const;
```

4 *Returns:* `!(*this == rhs)`.

```
bool before(const type_info& rhs) const;
```

5 *Effects:* Compares the current object with `rhs`.

6 *Returns:* `true` if `*this` precedes `rhs` in the implementation's collation order.

```
size_t hash_code() const throw();
```

7 *Returns:* an unspecified value, except that within a single execution of the program, it shall return the same value for any two `type_info` objects which compare equal.

8 *Remark:* an implementation should return different values for two `type_info` objects which do not compare equal.

```
const char* name() const;
```

9 *Returns:* an implementation-defined NTBS.

10 *Remarks:* The message may be a null-terminated multibyte string (17.5.3.2.4.3), suitable for conversion and display as a `wstring` (21.2, 22.2.1.4)

18.6.2 Class `type_index` [type.index]

18.6.2.1 `type_index` overview [type.index.overview]

```
namespace std {
  class type_index {
  public:
    type_index(const type_info& rhs);
    bool operator==(const type_index& rhs) const;
    bool operator!=(const type_index& rhs) const;
    bool operator< (const type_index& rhs) const;
    bool operator<= (const type_index& rhs) const;
    bool operator> (const type_index& rhs) const;
    bool operator>= (const type_index& rhs) const;
    size_t hash_code() const;
    const char* name() const;
  private:
    const type_info* target;     // exposition only
    // Note that the use of a pointer here, rather than a reference,
    // means that the default copy constructor and assignment
    // operator will be provided and work as expected.
  };
}
```

1 The class `type_index` provides a simple wrapper for `type_info` which can be used as an index type in associative containers (23.3) and in unordered associative containers (23.4).

18.6.2.2 `type_index` members [type.index.members]

`type_index(const type_info& rhs);`

1 *Effects:* constructs a `type_index` object, the equivalent of `target = &rhs`.

`bool operator==(const type_index& rhs) const;`

2 *Returns:* `*target == *rhs.target`

`bool operator!=(const type_index& rhs) const;`

3 *Returns:* `*target != *rhs.target`

`bool operator<(const type_index& rhs) const;`

4 *Returns:* `target->before(*rhs.target)`

`bool operator<=(const type_index& rhs) const;`

5 *Returns:* `!rhs.target->before(*target)`

`bool operator>(const type_index& rhs) const;`

6 *Returns:* `rhs.target->before(*target)`

`bool operator>=(const type_index& rhs) const;`

7 *Returns:* `!target->before(*rhs.target)`

```
size_t hash_code() const;
```

8 *Returns:* `target->hash_code()`

```
const char* name() const;
```

9 *Returns:* `target->name()`

18.6.2.3 Template specialization `hash<type_index>` [type.index.templ]

```
size_t operator()(type_index index) const;
```

1 *Returns:* `index.hash_code()`

18.6.3 Class `bad_cast` [bad.cast]

```
namespace std {
  class bad_cast : public exception {
  public:
    bad_cast() throw();
    bad_cast(const bad_cast&) throw();
    bad_cast& operator=(const bad_cast&) throw();
    virtual const char* what() const throw();
  };
}
```

1 The class `bad_cast` defines the type of objects thrown as exceptions by the implementation to report the execution of an invalid *dynamic-cast* expression (5.2.7).

```
bad_cast() throw();
```

2 *Effects:* Constructs an object of class `bad_cast`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_cast(const bad_cast&) throw();
bad_cast& operator=(const bad_cast&) throw();
```

4 *Effects:* Copies an object of class `bad_cast`.

```
virtual const char* what() const throw();
```

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.5.3.2.4.3), suitable for conversion and display as a `wstring` (21.2, 22.2.1.4)

18.6.4 Class `bad_typeid` [bad.typeid]

```
namespace std {
  class bad_typeid : public exception {
  public:
    bad_typeid() throw();
    bad_typeid(const bad_typeid&) throw();
    bad_typeid& operator=(const bad_typeid&) throw();
    virtual const char* what() const throw();
```

```
  };
}
```

1 The class `bad_typeid` defines the type of objects thrown as exceptions by the implementation to report a null pointer in a *typeid* expression (5.2.8).

```
bad_typeid() throw();
```

2 *Effects:* Constructs an object of class `bad_typeid`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_typeid(const bad_typeid&) throw();
bad_typeid& operator=(const bad_typeid&) throw();
```

4 *Effects:* Copies an object of class `bad_typeid`.

```
virtual const char* what() const throw();
```

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.5.3.2.4.3), suitable for conversion and display as a `wstring` (21.2, 22.2.1.4)

18.7 Exception handling [support.exception]

1 The header `<exception>` defines several types and functions related to the handling of exceptions in a C++ program.

Header `<exception>` synopsis

```
namespace std {
  class exception;
  class bad_exception;
  class nested_exception;

  typedef void (*unexpected_handler)();
  unexpected_handler set_unexpected(unexpected_handler f) throw();
  void unexpected();

  typedef void (*terminate_handler)();
  terminate_handler set_terminate(terminate_handler f) throw();
  void terminate();

  bool uncaught_exception() throw();

  typedef unspecified  exception_ptr;

  exception_ptr current_exception();
  void rethrow_exception(exception_ptr p);
  template<class E> exception_ptr copy_exception(E e);

  template <class T> void throw_with_nested(T&& t); // [[noreturn]]
  template <class E> void rethrow_if_nested(const E& e);
}
```

See also: 15.5.

18.7.1 Class `exception` [exception]

```
namespace std {
  class exception {
  public:
    exception() throw();
    exception(const exception&) throw();
    exception& operator=(const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
  };
}
```

1 The class `exception` defines the base class for the types of objects thrown as exceptions by C++ standard library components, and certain expressions, to report errors detected during program execution.

```
exception() throw();
```

2 *Effects:* Constructs an object of class `exception`.

3 *Remarks:* Does not throw any exceptions.

```
exception(const exception&) throw();
exception& operator=(const exception&) throw();
```

4 *Effects:* Copies an `exception` object.

5 *Remarks:* The effects of calling `what()` after assignment are implementation-defined.

```
virtual ~exception() throw();
```

6 *Effects:* Destroys an object of class `exception`.

7 *Remarks:* Does not throw any exceptions.

```
virtual const char* what() const throw();
```

8 *Returns:* An implementation-defined NTBS.

9 *Remarks:* The message may be a null-terminated multibyte string (17.5.3.2.4.3), suitable for conversion and display as a `wstring` (21.2, 22.2.1.4). The return value remains valid until the exception object from which it is obtained is destroyed or a non-`const` member function of the exception object is called.

18.7.2 Violating *exception-specification*s [exception.unexpected]

18.7.2.1 Class `bad_exception` [bad.exception]

```
namespace std {
  class bad_exception : public exception {
  public:
    bad_exception() throw();
    bad_exception(const bad_exception&) throw();
    bad_exception& operator=(const bad_exception&) throw();
    virtual const char* what() const throw();
  };
}
```

1 The class `bad_exception` defines the type of objects thrown as described in (15.5.2).

```
bad_exception() throw();
```

2 *Effects:* Constructs an object of class `bad_exception`.

3 *Remarks:* The result of calling `what()` on the newly constructed object is implementation-defined.

```
bad_exception(const bad_exception&) throw();
bad_exception& operator=(const bad_exception&) throw();
```

4 *Effects:* Copies an object of class `bad_exception`.

```
virtual const char* what() const throw();
```

5 *Returns:* An implementation-defined NTBS.

6 *Remarks:* The message may be a null-terminated multibyte string (17.5.3.2.4.3), suitable for conversion and display as a `wstring` (21.2, 22.2.1.4).

18.7.2.2 Type `unexpected_handler` [unexpected.handler]

```
typedef void (*unexpected_handler)();
```

1 The type of a *handler function* to be called by `unexpected()` when a function attempts to throw an exception not listed in its *exception-specification*.

2 *Required behavior:* An `unexpected_handler` shall not return. See also 15.5.2.

3 *Default behavior:* The implementation's default `unexpected_handler` calls `terminate()`.

18.7.2.3 `set_unexpected` [set.unexpected]

```
unexpected_handler set_unexpected(unexpected_handler f) throw();
```

1 *Effects:* Establishes the function designated by `f` as the current `unexpected_handler`.

2 *Requires:* `f` shall not be a null pointer.

3 *Returns:* The previous `unexpected_handler`.

18.7.2.4 `unexpected` [unexpected]

```
void unexpected();
```

1 Called by the implementation when a function exits via an exception not allowed by its *exception-specification* (15.5.2). May also be called directly by the program.

2 *Effects:* Calls the `unexpected_handler` function in effect immediately after evaluating the *throw-expression* (18.7.2.2), if called by the implementation, or calls the current `unexpected_handler`, if called by the program.

18.7.3 Abnormal termination [exception.terminate]

18.7.3.1 Type `terminate_handler` [terminate.handler]

```
typedef void (*terminate_handler)();
```

1 The type of a *handler function* to be called by `terminate()` when terminating exception processing.

2 *Required behavior:* A `terminate_handler` shall terminate execution of the program without returning to the caller.

3 *Default behavior:* The implementation's default `terminate_handler` calls `abort()`.

18.7.3.2 `set_terminate` [set.terminate]

```
terminate_handler set_terminate(terminate_handler f) throw();
```

1 *Effects:* Establishes the function designated by `f` as the current handler function for terminating exception processing.

2 *Requires:* `f` shall not be a null pointer.

3 *Returns:* The previous `terminate_handler`.

18.7.3.3 `terminate` [terminate]

```
void terminate();
```

1 Called by the implementation when exception handling must be abandoned for any of several reasons (15.5.1). May also be called directly by the program.

2 *Effects:* Calls the `terminate_handler` function in effect immediately after evaluating the *throw-expression* (18.7.3.1), if called by the implementation, or calls the current `terminate_handler` function, if called by the program.

18.7.4 `uncaught_exception` [uncaught]

```
bool uncaught_exception() throw();
```

1 *Returns:* `true` after completing evaluation of a *throw-expression* until either completing initialization of the *exception-declaration* in the matching handler or entering `unexpected()` due to the throw; or after entering `terminate()` for any reason other than an explicit call to `terminate()`. [*Note:* This includes stack unwinding (15.2). — *end note*]

2 *Remarks:* When `uncaught_exception()` returns `true`, throwing an exception can result in a call of `terminate()` (15.5.1).

18.7.5 Exception Propagation [propagation]

```
typedef unspecified  exception_ptr;
```

1 The type exception_ptr can be used to refer to an exception object.

2 `exception_ptr` shall be DefaultConstructible, CopyConstructible, Assignable and EqualityComparable. `exception_ptr`'s operations shall not throw exceptions.

3 Two objects of type `exception_ptr` are equivalent and compare equal if and only if they refer to the same exception.

4 The default constructor of `exception_ptr` produces the null value of the type. The null value is equivalent only to itself.

5 An object of type `exception_ptr` can be compared for equality with a null pointer constant and assigned a null pointer constant. The effect shall be as if `exception_ptr()` had been used in place of the null pointer constant.

6 [*Note:* An implementation might use a reference-counted smart pointer as `exception_ptr`. — *end note*]

```
exception_ptr current_exception();
```

7 *Returns:* An `exception_ptr` object that refers to the currently handled exception (15.3) or a copy of the currently handled exception, or a null `exception_ptr` object if no exception is being handled. The referenced object shall remain valid at least as long as there is an `exception_ptr` object that refers to it. If the function needs to allocate memory and the attempt fails, it returns an `exception_ptr` object that refers to an instance of `bad_alloc`. It is unspecified whether the return values of two successive calls to `current_exception` refer to the same exception object. [*Note:* that is, it is unspecified whether `current_exception` creates a new copy each time it is called. — *end note*] If the attempt to copy the current exception object throws an exception, the function returns an `exception_ptr` object that refers to the thrown exception or, if this is not possible, to an instance of `bad_exception`. [*Note:* The copy constructor of the thrown exception may also fail, so the implementation is allowed to substitute a `bad_exception` object to avoid infinite recursion. — *end note*]

8 *Throws:* nothing.

```
void rethrow_exception(exception_ptr p);
```

9 *Requires:* `p` shall not be a null pointer.

10 *Throws:* the exception object to which `p` refers.

```
template<class E> exception_ptr copy_exception(E e);
```

11 *Effects:* as if

```
try {
  throw e;
} catch(...) {
  return current_exception();
}
```

12 [*Note:* this function is provided for convenience and efficiency reasons. — *end note*]

18.7.6 `nested_exception` [except.nested]

```
namespace std {
  class nested_exception {
  public:
    nested_exception() throw();
    nested_exception(const nested_exception&) throw() = default;
    nested_exception& operator=(const nested_exception&) throw() = default;
    virtual ~nested_exception() = default;

    // access functions
    void rethrow_nested() const; // [[noreturn]]
    exception_ptr nested_ptr() const;
  };

  template<class T> void throw_with_nested(T&& t); // [[noreturn]]
```

```
  template <class E> void rethrow_if_nested(const E& e);
}
```

1 The class `nested_exception` is designed for use as a mixin through multiple inheritance. It captures the currently handled exception and stores it for later use.

2 [*Note:* `nested_exception` has a virtual destructor to make it a polymorphic class. Its presence can be tested for with `dynamic_cast`. — *end note*]

```
nested_exception() throw();
```

3 *Effects:* The constructor calls `current_exception()` and stores the returned value.

```
void rethrow_nested() const; // [[noreturn]]
```

4 *Throws:* the stored exception captured by this `nested_exception` object.

```
exception_ptr nested_ptr() const;
```

5 *Returns:* the stored exception captured by this `nested_exception` object.

```
template <class T> void throw_with_nested(T&& t); // [[noreturn]]
```

6 *Requires:* `T` shall be `CopyConstructible`.

7 *Throws:* If `T` is a non-union class type not derived from `nested_exception`, an exception of unspecified type that is publicly derived from both `T` and `nested_exception`, otherwise `t`.

```
template <class E> void rethrow_if_nested(const E& e);
```

8 *Effects:* Calls `e.rethrow_nested()` only if `e` is publicly derived from `nested_exception`.

18.8 Initializer lists [support.initlist]

1 The header `<initializer_list>` defines one type.

Header `<initializer_list>` synopsis

```
namespace std {
  template<ObjectType E> class initializer_list {
  public:
    typedef E value_type;
    typedef const E& reference;
    typedef const E& const_reference;
    typedef size_t size_type;

    typedef const E* iterator;
    typedef const E* const_iterator;

    initializer_list();

    size_t size() const;       // number of elements
    const E* begin() const;    // first element
    const E* end() const;      // one past the last element
  };

  template<typename T>
  concept_map Range<initializer_list<T> > see below;
```

```
  template<typename T>
  concept_map Range<const initializer_list<T> > see below;
}
```

2 An object of type `initializer_list<E>` provides access to an array of objects of type `const E`. [*Note:* A pair of pointers or a pointer plus a length would be obvious representations for `initializer_list`. `initializer_list` is used to implement initializer lists as specified in 8.5.4. Copying an initializer list does not copy the underlying elements. — *end note*]

18.8.1 Initializer list constructors [support.initlist.cons]

```
initializer_list();
```

1 *Effects:* constructs an empty `initializer_list` object.

2 *Postcondition:* `size() == 0`

3 *Throws:* nothing.

18.8.2 Initializer list access [support.initlist.access]

```
const E* begin() const;
```

1 *Returns:* a pointer to the beginning of the array. If `size() == 0` the values of `begin()` and `end()` are unspecified but they shall be identical.

2 *Throws:* nothing.

```
const E* end() const;
```

3 *Returns:* `begin() + size()`

4 *Throws:* nothing.

```
size_t size() const;
```

5 *Returns:* the number of elements in the array.

6 *Throws:* nothing.

18.8.3 Initializer list concept maps [support.initlist.concept]

```
template<typename T>
concept_map Range<initializer_list<T> > {
  typedef const T* iterator;

  iterator begin(initializer_list<T> r) { return r.begin(); }
  iterator end(initializer_list<T> r) { return r.end(); }
}

template<typename T>
concept_map Range<const initializer_list<T> > {
  typedef const T* iterator;

  iterator begin(initializer_list<T> r) { return r.begin(); }
  iterator end(initializer_list<T> r) { return r.end(); }
}
```

1 *Note:* these concept maps adapt initializer lists to the `Range` concept.

18.9 Other runtime support [support.runtime]

1 Headers `<cstdarg>` (variable arguments), `<csetjmp>` (nonlocal jumps), `<ctime>` (system clock `clock()`, `time()`), `<csignal>` (signal handling), `<cstdlib>` (runtime environment `getenv()`, `system()`), and `<cstdbool>` (`__bool_true_false_are_defined`).

Table 21 — Header `<cstdarg>` synopsis

Type	Name(s)			
Macros:	`va_arg`	`va_end`	`va_start`	`va_copy`
Type:	`va_list`			

Table 22 — Header `<csetjmp>` synopsis

Type	Name(s)
Macro:	`setjmp`
Type:	`jmp_buf`
Function:	`longjmp`

Table 23 — Header `<ctime>` synopsis

Type	Name(s)
Macro:	`CLOCKS_PER_SEC`
Type:	`clock_t`
Function:	`clock`

Table 24 — Header `<csignal>` synopsis

Type	Name(s)			
Macros:	`SIGABRT`	`SIGILL`	`SIGSEGV`	`SIG_DFL`
`SIG_IGN`	`SIGFPE`	`SIGINT`	`SIGTERM`	`SIG_ERR`
Type:	`sig_atomic_t`			
Functions:	`raise`	`signal`		

2 The contents of these headers are the same as the Standard C library headers `<stdarg.h>`, `<setjmp.h>`, `<time.h>`, `<signal.h>`, and `<stdlib.h>` respectively, with the following changes:

3 The restrictions that ISO C places on the second parameter to the `va_start()` macro in header `<stdarg.h>` are different in this International Standard. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`).[224] If the parameter `parmN` is declared with a function, array, or reference type, or with a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.

See also: ISO C 4.8.1.1.

224) Note that `va_start` is required to work as specified even if unary `operator&` is overloaded for the type of `parmN`.

Table 25 — Header <cstdlib> synopsis

Type	Name(s)
Functions:	getenv system

Table 26 — Header <cstdbool> synopsis

Type	Name(s)
Macro:	__bool_true_false_are_defined

4 The function signature longjmp(jmp_buf jbuf, int val) has more restricted behavior in this International Standard. A setjmp/longjmp call pair has undefined behavior if replacing the setjmp and longjmp by catch and throw would destroy any automatic objects.

See also: ISO C 7.10.4, 7.8, 7.6, 7.12.

5 The header <cstdbool> and the header <stdbool.h> shall not define macros named bool, true, and false.

6 The common subset of the C and C++ languages consists of all declarations, definitions, and expressions that may appear in a well formed C++ program and also in a conforming C program. A POF ("plain old function") is a function that uses only features from this common subset, and that does not directly or indirectly use any function that is not a POF, except that it may use functions defined in Clause 29 that are not member functions. All signal handlers shall have C linkage. A POF that could be used as a signal handler in a conforming C program does not produce undefined behavior when used as a signal handler in a C++ program. The behavior of any other function used as a signal handler in a C++ program is implementation-defined.[225]

225) In particular, a signal handler using exception handling is very likely to have problems. Also, invoking std::exit may cause destruction of objects, including those of the standard library implementation, which, in general, yields undefined behavior in a signal handler (see 1.9).

19 Diagnostics library [diagnostics]

1 This Clause describes components that C++ programs may use to detect and report error conditions.

2 The following subclauses describe components for reporting several kinds of exceptional conditions, documenting program assertions, and a global variable for error number codes, as summarized in Table 27.

Table 27 — Diagnostics library summary

	Subclause	Header(s)
19.1	Exception classes	`<stdexcept>`
19.2	Assertions	`<cassert>`
19.3	Error numbers	`<cerrno>`
19.4	System error support	`<system_error>`

19.1 Exception classes [std.exceptions]

1 The Standard C++ library provides classes to be used to report certain errors (17.6.5.10) in C++ programs. In the error model reflected in these classes, errors are divided into two broad categories: *logic* errors and *runtime* errors.

2 The distinguishing characteristic of logic errors is that they are due to errors in the internal logic of the program. In theory, they are preventable.

3 By contrast, runtime errors are due to events beyond the scope of the program. They cannot be easily predicted in advance. The header `<stdexcept>` defines several types of predefined exceptions for reporting errors in a C++ program. These exceptions are related by inheritance.

Header `<stdexcept>` synopsis

```
namespace std {
  class logic_error;
    class domain_error;
    class invalid_argument;
    class length_error;
    class out_of_range;
  class runtime_error;
    class range_error;
    class overflow_error;
    class underflow_error;
}
```

19.1.1 Class `logic_error` [logic.error]

```
namespace std {
  class logic_error : public exception {
  public:
    explicit logic_error(const string& what_arg);
    explicit logic_error(const char* what_arg);
  };
}
```

1 The class `logic_error` defines the type of objects thrown as exceptions to report errors presumably detectable before the program executes, such as violations of logical preconditions or class invariants.

```
logic_error(const string& what_arg);
```

2 *Effects:* Constructs an object of class `logic_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
logic_error(const char* what_arg);
```

4 *Effects:* Constructs an object of class `logic_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.2 Class `domain_error` [domain.error]

```
namespace std {
  class domain_error : public logic_error {
  public:
    explicit domain_error(const string& what_arg);
    explicit domain_error(const char* what_arg);
  };
}
```

1 The class `domain_error` defines the type of objects thrown as exceptions by the implementation to report domain errors.

```
domain_error(const string& what_arg);
```

2 *Effects:* Constructs an object of class `domain_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
domain_error(const char* what_arg);
```

4 *Effects:* Constructs an object of class `domain_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.3 Class `invalid_argument` [invalid.argument]

```
namespace std {
  class invalid_argument : public logic_error {
  public:
    explicit invalid_argument(const string& what_arg);
    explicit invalid_argument(const char* what_arg);
  };
}
```

1 The class `invalid_argument` defines the type of objects thrown as exceptions to report an invalid argument.

```
invalid_argument(const string& what_arg);
```

2 *Effects:* Constructs an object of class `invalid_argument`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
invalid_argument(const char* what_arg);
```

4 *Effects:* Constructs an object of class `invalid_argument`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.4 Class `length_error` [length.error]

```
namespace std {
  class length_error : public logic_error {
  public:
    explicit length_error(const string& what_arg);
    explicit length_error(const char* what_arg);
  };
}
```

1 The class `length_error` defines the type of objects thrown as exceptions to report an attempt to produce an object whose length exceeds its maximum allowable size.

```
length_error(const string& what_arg);
```

2 *Effects:* Constructs an object of class `length_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
length_error(const char* what_arg);
```

4 *Effects:* Constructs an object of class `length_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.5 Class `out_of_range` [out.of.range]

```
namespace std {
  class out_of_range : public logic_error {
  public:
    explicit out_of_range(const string& what_arg);
    explicit out_of_range(const char* what_arg);
  };
}
```

1 The class `out_of_range` defines the type of objects thrown as exceptions to report an argument value not in its expected range.

```
out_of_range(const string& what_arg);
```

2 *Effects:* Constructs an object of class `out_of_range`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
out_of_range(const char* what_arg);
```

4 *Effects:* Constructs an object of class `out_of_range`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.6 Class `runtime_error` [runtime.error]

```
namespace std {
  class runtime_error : public exception {
  public:
    explicit runtime_error(const string& what_arg);
    explicit runtime_error(const char* what_arg);
  };
}
```

1 The class `runtime_error` defines the type of objects thrown as exceptions to report errors presumably detectable only when the program executes.

`runtime_error(const string& what_arg);`

2 *Effects:* Constructs an object of class `runtime_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`runtime_error(const char* what_arg);`

4 *Effects:* Constructs an object of class `runtime_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.7 Class `range_error` [range.error]

```
namespace std {
  class range_error : public runtime_error {
  public:
    explicit range_error(const string& what_arg);
    explicit range_error(const char* what_arg);
  };
}
```

1 The class `range_error` defines the type of objects thrown as exceptions to report range errors in internal computations.

`range_error(const string& what_arg);`

2 *Effects:* Constructs an object of class `range_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

`range_error(const char* what_arg);`

4 *Effects:* Constructs an object of class `range_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.8 Class `overflow_error` [overflow.error]

```
namespace std {
  class overflow_error : public runtime_error {
  public:
    explicit overflow_error(const string& what_arg);
    explicit overflow_error(const char* what_arg);
  };
```

```
}
```

1 The class `overflow_error` defines the type of objects thrown as exceptions to report an arithmetic overflow error.

```
overflow_error(const string& what_arg);
```

2 *Effects:* Constructs an object of class `overflow_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
overflow_error(const char* what_arg);
```

4 *Effects:* Constructs an object of class `overflow_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.1.9 Class `underflow_error` [underflow.error]

```
namespace std {
  class underflow_error : public runtime_error {
  public:
    explicit underflow_error(const string& what_arg);
    explicit underflow_error(const char* what_arg);
  };
}
```

1 The class `underflow_error` defines the type of objects thrown as exceptions to report an arithmetic underflow error.

```
underflow_error(const string& what_arg);
```

2 *Effects:* Constructs an object of class `underflow_error`.

3 *Postcondition:* `strcmp(what(), what_arg.c_str()) == 0`.

```
underflow_error(const char* what_arg);
```

4 *Effects:* Constructs an object of class `underflow_error`.

5 *Postcondition:* `strcmp(what(), what_arg) == 0`.

19.2 Assertions [assertions]

1 The header `<cassert>`, described in (Table 28), provides a macro for documenting C++ program assertions and a mechanism for disabling the assertion checks.

Table 28 — Header `<cassert>` synopsis

Type	Name(s)
Macro:	`assert`

2 The contents are the same as the Standard C library header `<assert.h>`.

See also: ISO C 7.2.

19.3 Error numbers [errno]

1

The header <cerrno> is described in Table 29. Its contents are the same as the POSIX header <errno.h>, except that errno shall be defined as a macro. [*Note:* The intent is to remain in close alignment with the POSIX standard. — *end note*] A separate errno value shall be provided for each thread.

Table 29 — Header <cerrno> synopsis

Type	Name(s)				
Macros:	ECONNREFUSED	EIO	ENODEV	ENOTEMPTY	ERANGE
E2BIG	ECONNRESET	EISCONN	ENOENT	ENOTRECOVERABLE	EROFS
EACCES	EDEADLK	EISDIR	ENOEXEC	ENOTSOCK	ESPIPE
EADDRINUSE	EDESTADDRREQ	ELOOP	ENOLCK	ENOTSUP	ESRCH
EADDRNOTAVAIL	EDOM	EMFILE	ENOLINK	ENOTTY	ETIME
EAFNOSUPPORT	EEXIST	EMLINK	ENOMEM	ENXIO	ETIMEDOUT
EAGAIN	EFAULT	EMSGSIZE	ENOMSG	EOPNOTSUPP	ETXTBSY
EALREADY	EFBIG	ENAMETOOLONG	ENOPROTOOPT	EOVERFLOW	EWOULDBLOCK
EBADF	EHOSTUNREACH	ENETDOWN	ENOSPC	EOWNERDEAD	EXDEV
EBADMSG	EIDRM	ENETRESET	ENOSR	EPERM	errno
EBUSY	EILSEQ	ENETUNREACH	ENOSTR	EPIPE	
ECANCELED	EINPROGRESS	ENFILE	ENOSYS	EPROTO	
ECHILD	EINTR	ENOBUFS	ENOTCONN	EPROTONOSUPPORT	
ECONNABORTED	EINVAL	ENODATA	ENOTDIR	EPROTOTYPE	

See also: ISO C 7.1.4, 7.2, Amendment 1 4.3.

19.4 System error support [syserr]

1 This subclause describes components that the standard library and C++ programs may use to report error conditions originating from the operating system or other low-level application program interfaces.

2 Components described in this subclause shall not change the value of errno (19.3). Implementations are encouraged but not required to leave unchanged the error states provided by other libraries.

Header <system_error> synopsis

```
namespace std {
  class error_category;
  class error_code;
  class error_condition;
  class system_error;

  concept ErrorCodeEnum<typename T> see below;
  concept ErrorConditionEnum<typename T> see below;

  enum class errc {
    address_family_not_supported,       // EAFNOSUPPORT
    address_in_use,                     // EADDRINUSE
    address_not_available,              // EADDRNOTAVAIL
    already_connected,                  // EISCONN
    argument_list_too_long,             // E2BIG
    argument_out_of_domain,             // EDOM
```

```
    bad_address,                        // EFAULT
    bad_file_descriptor,                // EBADF
    bad_message,                        // EBADMSG
    broken_pipe,                        // EPIPE
    connection_aborted,                 // ECONNABORTED
    connection_already_in_progress,     // EALREADY
    connection_refused,                 // ECONNREFUSED
    connection_reset,                   // ECONNRESET
    cross_device_link,                  // EXDEV
    destination_address_required,       // EDESTADDRREQ
    device_or_resource_busy,            // EBUSY
    directory_not_empty,                // ENOTEMPTY
    executable_format_error,            // ENOEXEC
    file_exists,                        // EEXIST
    file_too_large,                     // EFBIG
    filename_too_long,                  // ENAMETOOLONG
    function_not_supported,             // ENOSYS
    host_unreachable,                   // EHOSTUNREACH
    identifier_removed,                 // EIDRM
    illegal_byte_sequence,              // EILSEQ
    inappropriate_io_control_operation, // ENOTTY
    interrupted,                        // EINTR
    invalid_argument,                   // EINVAL
    invalid_seek,                       // ESPIPE
    io_error,                           // EIO
    is_a_directory,                     // EISDIR
    message_size,                       // EMSGSIZE
    network_down,                       // ENETDOWN
    network_reset,                      // ENETRESET
    network_unreachable,                // ENETUNREACH
    no_buffer_space,                    // ENOBUFS
    no_child_process,                   // ECHILD
    no_link,                            // ENOLINK
    no_lock_available,                  // ENOLCK
    no_message_available,               // ENODATA
    no_message,                         // ENOMSG
    no_protocol_option,                 // ENOPROTOOPT
    no_space_on_device,                 // ENOSPC
    no_stream_resources,                // ENOSR
    no_such_device_or_address,          // ENXIO
    no_such_device,                     // ENODEV
    no_such_file_or_directory,          // ENOENT
    no_such_process,                    // ESRCH
    not_a_directory,                    // ENOTDIR
    not_a_socket,                       // ENOTSOCK
    not_a_stream,                       // ENOSTR
    not_connected,                      // ENOTCONN
    not_enough_memory,                  // ENOMEM
    not_supported,                      // ENOTSUP
    operation_canceled,                 // ECANCELED
    operation_in_progress,              // EINPROGRESS
    operation_not_permitted,            // EPERM
    operation_not_supported,            // EOPNOTSUPP
    operation_would_block,              // EWOULDBLOCK
    owner_dead,                         // EOWNERDEAD
```

```
    permission_denied,                  // EACCES
    protocol_error,                     // EPROTO
    protocol_not_supported,             // EPROTONOSUPPORT
    read_only_file_system,              // EROFS
    resource_deadlock_would_occur,      // EDEADLK
    resource_unavailable_try_again,     // EAGAIN
    result_out_of_range,                // ERANGE
    state_not_recoverable,              // ENOTRECOVERABLE
    stream_timeout,                     // ETIME
    text_file_busy,                     // ETXTBSY
    timed_out,                          // ETIMEDOUT
    too_many_files_open_in_system,      // ENFILE
    too_many_files_open,                // EMFILE
    too_many_links,                     // EMLINK
    too_many_symbolic_link_levels,      // ELOOP
    value_too_large,                    // EOVERFLOW
    wrong_protocol_type,                // EPROTOTYPE
  };

  concept_map ErrorConditionEnum<errc> { };

  error_code make_error_code(errc e);
  error_condition make_error_condition(errc e);

  // 19.4.4 Comparison operators:
  bool operator==(const error_code& lhs, const error_code& rhs);
  bool operator==(const error_code& lhs, const error_condition& rhs);
  bool operator==(const error_condition& lhs, const error_code& rhs);
  bool operator==(const error_condition& lhs, const error_condition& rhs);
  bool operator!=(const error_code& lhs, const error_code& rhs);
  bool operator!=(const error_code& lhs, const error_condition& rhs);
  bool operator!=(const error_condition& lhs, const error_code& rhs);
  bool operator!=(const error_condition& lhs, const error_condition& rhs);
} // namespace std
```

3 The value of each `enum errc` constant shall be the same as the value of the `<cerrno>` macro shown in the above synopsis. Whether or not the `<system_error>` implementation exposes the `<cerrno>` macros is unspecified.

19.4.1 Class `error_category` [syserr.errcat]

19.4.1.1 Class `error_category` overview [syserr.errcat.overview]

1 The class `error_category` serves as a base class for types used to identify the source and encoding of a particular category of error code. Classes may be derived from `error_category` to support categories of errors in addition to those defined in this International Standard. Such classes shall behave as specified in this subclause. [*Note:* `error_category` objects are passed by reference, and two such objects are equal if they have the same address. This means that applications using custom `error_category` types should create a single object of each such type. — *end note*]

```
namespace std {
  class error_category {
  public:
    virtual ~error_category();
    error_category(const error_category&) = delete;
```

```
    error_category& operator=(const error_category&) = delete;
    virtual const char* name() const = 0;
    virtual error_condition default_error_condition(int ev) const;
    virtual bool equivalent(int code, const error_condition& condition) const;
    virtual bool equivalent(const error_code& code, int condition) const;
    virtual string message(int ev) const = 0;

    bool operator==(const error_category& rhs) const;
    bool operator!=(const error_category& rhs) const;
    bool operator<(const error_category& rhs) const;
  };

  const error_category& get_generic_category();
  const error_category& get_system_category();

  static const error_category& generic_category = get_generic_category();
  static const error_category& system_category = get_system_category();
}  // namespace std
```

19.4.1.2 Class `error_category` virtual members [syserr.errcat.virtuals]

```
virtual const char* name() const = 0;
```

1 *Returns:* A string naming the error category.

2 *Throws:* Nothing.

```
virtual error_condition default_error_condition(int ev) const;
```

3 *Returns:* `error_condition(ev, *this)`.

4 *Throws:* Nothing.

```
virtual bool equivalent(int code, const error_condition& condition) const;
```

5 *Returns:* `default_error_condition(code) == condition`.

6 *Throws:* Nothing.

```
virtual bool equivalent(const error_code& code, int condition) const;
```

7 *Returns:* `*this == code.category() && code.value() == condition`.

8 *Throws:* Nothing.

```
virtual string message(int ev) const = 0;
```

9 *Returns:* A string that describes the error condition denoted by `ev`.

19.4.1.3 Class `error_category` non-virtual members [syserr.errcat.nonvirtuals]

```
bool operator==(const error_category& rhs) const;
```

1 *Returns:* `this == &rhs`.

```
bool operator!=(const error_category& rhs) const;
```

2 *Returns:* `!(*this == rhs)`.

```
bool operator<(const error_category& rhs) const;
```

3 *Returns:* `less<const error_category*>()(this, &rhs)`.

[*Note:* `less` (20.6.8) provides a total ordering for pointers. — *end note*]

4 *Throws:* Nothing.

19.4.1.4 Program defined classes derived from `error_category` [syserr.errcat.derived]

```
virtual const char *name() const = 0;
```

1 *Returns:* a string naming the error category.

2 *Throws:* Nothing.

```
virtual error_condition default_error_condition(int ev) const;
```

3 *Returns:* An object of type `error_condition` that corresponds to `ev`.

4 *Throws:* Nothing.

```
virtual bool equivalent(int code, const error_condition& condition) const;
```

5 *Returns:* `true` if, for the category of error represented by `*this`, `code` is considered equivalent to `condition`; otherwise, `false`.

6 *Throws:* Nothing.

```
virtual bool equivalent(const error_code& code, int condition) const;
```

7 *Returns:* `true` if, for the category of error represented by `*this`, `code` is considered equivalent to `condition`; otherwise, `false`.

8 *Throws:* Nothing.

19.4.1.5 Error category objects [syserr.errcat.objects]

```
const error_category& get_generic_category();
```

1 *Returns:* A reference to an object of a type derived from class `error_category`.

2 *Remarks:* The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"generic"`.

```
const error_category& get_system_category();
```

3 *Returns:* A reference to an object of a type derived from class `error_category`.

4 *Remarks:* The object's `equivalent` virtual functions shall behave as specified for class `error_category`. The object's `name` virtual function shall return a pointer to the string `"system"`. The object's `default_error_condition` virtual function shall behave as follows:

If the argument `ev` corresponds to a POSIX `errno` value `posv`, the function shall return `error_condition(posv, generic_category)`. Otherwise, the function shall return `error_condition(ev, system_category)`. What constitutes correspondence for any given operating system is unspecified. [*Note:* The number of potential system error codes is large and unbounded, and some may not correspond to any POSIX `errno` value. Thus implementations are given latitude in determining correspondence. — *end note*]

19.4.2 Class `error_code` and concept `ErrorCodeEnum` [syserr.errcode]

19.4.2.1 Concept `ErrorCodeEnum` [syserr.errcodeenum]

```
namespace std {
  concept ErrorCodeEnum<typename T> : EnumerationType<T> {
    error_code make_error_code(T val);
  }
}
```

1 *Requires:* Function `make_error_code` shall return `error_code(static_cast<int>(val), cat)`, where `cat` is an `error_category` that describes type `T`.

2 *Remark:* Describes types to be used as an argument to function `make_error_code`.

19.4.2.2 Class `error_code` overview [syserr.errcode.overview]

1 The class `error_code` describes an object used to hold error code values, such as those originating from the operating system or other low-level application program interfaces. [*Note:* Class `error_code` is an adjunct to error reporting by exception. — *end note*]

```
namespace std {
  class error_code {
  public:
    // 19.4.2.3 constructors:
    error_code();
    error_code(int val, const error_category& cat);
    template <ErrorCodeEnum E>
      error_code(E e);

    // 19.4.2.4 modifiers:
    void assign(int val, const error_category& cat);
    template <ErrorCodeEnum E>
      error_code& operator=(E e);
    void clear();

    // 19.4.2.5 observers:
    int value() const;
    const error_category& category() const;
    error_condition default_error_condition() const;
    string message() const;
    explicit operator bool() const;

  private:
    int val_;                   // exposition only
    const error_category* cat_; // exposition only
  };

  // 19.4.2.6 non-member functions:
  bool operator<(const error_code& lhs, const error_code& rhs);

  template <class charT, class traits>
    basic_ostream<charT,traits>&
      operator<<(basic_ostream<charT,traits>& os, const error_code& ec);
}   // namespace std
```

19.4.2.3 Class `error_code` constructors [syserr.errcode.constructors]

```
error_code();
```

1 *Effects:* Constructs an object of type `error_code`.

2 *Postconditions:* `val_ == 0` and `cat_ == &system_category`.

3 *Throws:* Nothing.

```
error_code(int val, const error_category& cat);
```

4 *Effects:* Constructs an object of type `error_code`.

5 *Postconditions:* `val_ == val` and `cat_ == &cat`.

6 *Throws:* Nothing.

```
template <ErrorCodeEnum E>
  error_code(E e);
```

7 *Effects:* Constructs an object of type `error_code`.

8 *Postconditions:* `*this == make_error_code(e)`.

9 *Throws:* Nothing.

19.4.2.4 Class `error_code` modifiers [syserr.errcode.modifiers]

```
void assign(int val, const error_category& cat);
```

1 *Postconditions:* `val_ == val` and `cat_ == &cat`.

2 *Throws:* Nothing.

```
template <ErrorCodeEnum E>
  error_code& operator=(E e);
```

3 *Postconditions:* `*this == make_error_code(e)`.

4 *Returns:* `*this`.

5 *Throws:* Nothing.

```
void clear();
```

6 *Postconditions:* `value() == 0` and `category() == system_category`.

19.4.2.5 Class `error_code` observers [syserr.errcode.observers]

```
int value() const;
```

1 *Returns:* `val_`.

2 *Throws:* Nothing.

```
const error_category& category() const;
```

3 *Returns:* `*cat_`.

4 *Throws:* Nothing.

```
error_condition default_error_condition() const;
```

Returns: `category().default_error_condition(value())`.

Throws: Nothing.

```
string message() const;
```

Returns: `category().message(value())`.

```
explicit operator bool() const;
```

Returns: `value() != 0`.

Throws: Nothing.

19.4.2.6 Class `error_code` non-member functions [syserr.errcode.nonmembers]

```
error_code make_error_code(errc e);
```

Returns: `error_code(static_cast<int>(e), generic_category)`.

```
bool operator<(const error_code& lhs, const error_code& rhs);
```

Returns: `lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value()`.

Throws: Nothing.

```
template <class charT, class traits>
  basic_ostream<charT,traits>&
    operator<<(basic_ostream<charT,traits>& os, const error_code& ec);
```

Effects: `os << ec.category().name() << ':' << ec.value()`.

19.4.3 Class `error_condition` and concept `ErrorConditionEnum` [syserr.errcondition]

19.4.3.1 Concept `ErrorConditionEnum` [syserr.errcondenum]

```
namespace std {
  concept ErrorConditionEnum<typename T> : EnumerationType<T> {
    error_condition make_error_condition(T val);
  }
}
```

Requires: Function `make_error_condition` shall return `error_condition(static_cast<int>(val), cat)`, where `cat` is an `error_category` that describes type `T`.

Remark: Describes types to be used as an argument to function `make_error_condition`.

19.4.3.2 Class `error_condition` overview [syserr.errcondition.overview]

The class `error_condition` describes an object used to hold values identifying error conditions. [*Note:* `error_condition` values are portable abstractions, while `error_code` values (19.4.2) are implementation specific. — *end note*]

```
  namespace std {
    class error_condition {
    public:
```

```
    // 19.4.3.3 constructors:
    error_condition();
    error_condition(int val, const error_category& cat);
    template <ErrorConditionEnum E>
      error_condition(E e);

    // 19.4.3.4 modifiers:
    void assign(int val, const error_category& cat);
    template <ErrorConditionEnum E>
        error_condition& operator=(E e);
    void clear();

    // 19.4.3.5 observers:
    int value() const;
    const error_category& category() const;
    string message() const;
    explicit operator bool() const;

  private:
    int val_;                   // exposition only
    const error_category* cat_; // exposition only
  };

  // 19.4.3.6 non-member functions:
  bool operator<(const error_condition& lhs, const error_condition& rhs);
} // namespace std
```

19.4.3.3 Class `error_condition` constructors [syserr.errcondition.constructors]

```
error_condition();
```

1 *Effects:* Constructs an object of type `error_condition`.

2 *Postconditions:* `val_ == 0` and `cat_ == &generic_category`.

3 *Throws:* Nothing.

```
error_condition(int val, const error_category& cat);
```

4 *Effects:* Constructs an object of type `error_condition`.

5 *Postconditions:* `val_ == val` and `cat_ == &cat`.

6 *Throws:* Nothing.

```
template <ErrorConditionEnum E>
  error_condition(E e);
```

7 *Effects:* Constructs an object of type `error_condition`.

8 *Postcondition:* `*this == make_error_condition(e)`.

9 *Throws:* Nothing.

19.4.3.4 Class `error_condition` modifiers [syserr.errcondition.modifiers]

```
void assign(int val, const error_category& cat);
```

1 *Postconditions:* `val_ == val` and `cat_ == &cat`.

2 *Throws:* Nothing.

```
template <ErrorConditionEnum E>
    error_condition& operator=(E e);
```

3 *Postcondition:* `*this == make_error_condition(e)`.

4 *Throws:* Nothing.

```
void clear();
```

Postconditions: `value() == 0` and `category() == generic_category`.

19.4.3.5 Class `error_condition` observers [syserr.errcondition.observers]

```
int value() const;
```

1 *Returns:* `val_`.

2 *Throws:* Nothing.

```
const error_category& category() const;
```

3 *Returns:* `*cat_`.

4 *Throws:* Nothing.

```
string message() const;
```

5 *Returns:* `category().message(value())`.

```
explicit operator bool() const;
```

6 *Returns:* `value() != 0`.

7 *Throws:* Nothing.

19.4.3.6 Class `error_condition` non-member functions [syserr.errcondition.nonmembers]

```
error_condition make_error_condition(errc e);
```

Returns: `error_condition(static_cast<int>(e), generic_category)`.

```
bool operator<(const error_condition& lhs, const error_condition& rhs);
```

1 *Returns:* `lhs.category() < rhs.category() || lhs.category() == rhs.category() && lhs.value() < rhs.value()`.

2 *Throws:* Nothing.

19.4.4 Comparison operators [syserr.compare]

```
bool operator==(const error_code& lhs, const error_code& rhs);
```

1 *Returns:* `lhs.category() == rhs.category() && lhs.value() == rhs.value()`.

2 *Throws:* Nothing.

```
bool operator==(const error_code& lhs, const error_condition& rhs);
```

3 *Returns:* `lhs.category().equivalent(lhs.value(), rhs) || rhs.category().equivalent(lhs, rhs.value())`.

4 *Throws:* Nothing.

```
bool operator==(const error_condition& lhs, const error_code& rhs);
```

5 *Returns:* `rhs.category().equivalent(rhs.value(), lhs) || lhs.category.equivalent(rhs, lhs.value())`.

6 *Throws:* Nothing.

```
bool operator==(const error_condition& lhs, const error_condition& rhs);
```

7 *Returns:* `lhs.category() == rhs.category() && lhs.value() == rhs.value()`.

8 *Throws:* Nothing.

```
bool operator!=(const error_code& lhs, const error_code& rhs);
bool operator!=(const error_code& lhs, const error_condition& rhs);
bool operator!=(const error_condition& lhs, const error_code& rhs);
bool operator!=(const error_condition& lhs, const error_condition& rhs);
```

9 *Returns:* `!(lhs == rhs)`.

10 *Throws:* Nothing.

19.4.5 Class `system_error` [syserr.syserr]

19.4.5.1 Class `system_error` overview [syserr.syserr.overview]

1 The class `system_error` describes an exception object used to report error conditions that have an associated error code. Such error conditions typically originate from the operating system or other low-level application program interfaces.

2 [*Note:* If an error represents an out-of-memory condition, implementations are encouraged to throw an exception object of type `bad_alloc` 18.5.2.1 rather than `system_error`. — *end note*]

```
namespace std {
  class system_error : public runtime_error {
  public:
    system_error(error_code ec, const string& what_arg);
    system_error(error_code ec, const char* what_arg);
    system_error(error_code ec);
    system_error(int ev, const error_category& ecat,
        const string& what_arg);
    system_error(int ev, const error_category& ecat,
        const char* what_arg);
    system_error(int ev, const error_category& ecat);
    const error_code& code() const throw();
    const char* what() const throw();
  };
}   // namespace std
```

19.4.5.2 Class `system_error` members [syserr.syserr.members]

```
system_error(error_code ec, const string& what_arg);
```

1 *Effects:* Constructs an object of class `system_error`.

2 *Postconditions:* `code() == ec` and `strcmp(runtime_error::what(), what_arg.c_str()) == 0`.

```
system_error(error_code ec, const char* what_arg);
```

3 *Effects:* Constructs an object of class `system_error`.

4 *Postconditions:* `code() == ec` and `strcmp(runtime_error::what(), what_arg) == 0`.

```
system_error(error_code ec);
```

5 *Effects:* Constructs an object of class `system_error`.

6 *Postconditions:* `code() == ec` and `strcmp(runtime_error::what(), "") == 0`.

```
system_error(int ev, const error_category& ecat,
  const string& what_arg);
```

7 *Effects:* Constructs an object of class `system_error`.

8 *Postconditions:* `code() == error_code(ev, ecat)` and `strcmp(runtime_error::what(), what_arg.c_str()) == 0`.

```
system_error(int ev, const error_category& ecat,
  const char* what_arg);
```

9 *Effects:* Constructs an object of class `system_error`.

10 *Postconditions:* `code() == error_code(ev, ecat)` and `strcmp(runtime_error::what(), what_arg) == 0`.

```
system_error(int ev, const error_category& ecat);
```

11 *Effects:* Constructs an object of class `system_error`.

12 *Postconditions:* `code() == error_code(ev, ecat)` and `strcmp(runtime_error::what(), "") == 0`.

```
const error_code& code() const throw();
```

13 *Returns:* `ec` or `error_code(ev, ecat)`, from the constructor, as appropriate.

```
const char *what() const throw();
```

14 *Returns:* An NTBS incorporating `runtime_error::what()` and `code().message()`.

[*Note:* One possible implementation would be:

```
if (msg.empty()) {
  try {
    std::string tmp = runtime_error::what();
    if (code()) {
      if (!tmp.empty())
        tmp += ": ";
      tmp += code().message();
    }
    swap(msg, tmp);
  } catch(...) {
    return runtime_error::what();
  }
return msg.c_str();
```

— *end note*]

20 General utilities library [utilities]

1 This Clause describes components used by other elements of the C++ standard library. These components may also be used by C++ programs.

2 The following Clauses describe utility and allocator requirements, utility components, compile-time rational arithmetic, tuples, type traits templates, function objects, dynamic memory management utilities, and date/time utilities, as summarized in Table 30.

Table 30 — General utilities library summary

Subclause		Header(s)
20.1	Concepts	`<concepts>` `<memory_concepts>`
20.2	Utility components	`<utility>`
20.3	Compile-time rational arithmetic	`<ratio>`
20.4	Tuples	`<tuple>`
20.5	Type traits	`<type_traits>`
20.6	Function objects	`<functional>`

20.7	Memory	`<memory>`
		`<cstdlib>`
		`<cstring>`

20.8	Time utilities	`<chrono>`
20.9	Date and time functions	`<ctime>`

20.1 Concepts [utility.concepts]

1 This subclause describes concepts that specify requirements on template arguments used throughout the C++ Standard Library. Concepts whose name is prefixed with `Has` provide detection of a specific syntax (e.g., `HasConstructor`), but do not imply the semantics of the corresponding operation. Concepts whose name has the `able` or `ible` suffix (e.g., `Constructible`) require both a specific syntax and semantics of the associated operations. These semantic concepts refine the corresponding syntax-detection concepts, for example, the `Constructible` concept refines the `HasConstructor` concept.

Header `<concepts>` synopsis

```
namespace std {
  // 20.1.1, type transformations:
  auto concept IdentityOf<typename T> see below;
  auto concept RvalueOf<typename T> see below;
  template<typename T> concept_map RvalueOf<T&> see below;

  // 20.1.2, true:
  concept True<bool> { }
  concept_map True<true> { }

  // 20.1.3, operator concepts:
  auto concept HasPlus<typename T, typename U> see below;
```

```
auto concept HasMinus<typename T, typename U> see below;
auto concept HasMultiply<typename T, typename U> see below;
auto concept HasDivide<typename T, typename U> see below;
auto concept HasModulus<typename T, typename U> see below;
auto concept HasUnaryPlus<typename T> see below;
auto concept HasNegate<typename T> see below;
auto concept HasLess<typename T, typename U> see below;
auto concept HasGreater<typename T, typename U> see below;
auto concept HasLessEqual<typename T, typename U> see below;
auto concept HasGreaterEqual<typename T, typename U> see below;
auto concept HasEqualTo<typename T, typename U> see below;
auto concept HasNotEqualTo<typename T, typename U> see below;
auto concept HasLogicalAnd<typename T, typename U> see below;
auto concept HasLogicalOr<typename T, typename U> see below;
auto concept HasLogicalNot<typename T> see below;
auto concept HasBitAnd<typename T, typename U> see below;
auto concept HasBitOr<typename T, typename U> see below;
auto concept HasBitXor<typename T, typename U> see below;
auto concept HasComplement<typename T> see below;
auto concept HasLeftShift<typename T, typename U> see below;
auto concept HasRightShift<typename T, typename U> see below;
auto concept HasDereference<typename T> see below;
auto concept HasAddressOf<typename T> see below;
auto concept HasSubscript<typename T, typename U> see below;
auto concept Callable<typename F, typename... Args> see below;
auto concept HasAssign<typename T, typename U> see below;
auto concept HasPlusAssign<typename T, typename U> see below;
auto concept HasMinusAssign<typename T, typename U> see below;
auto concept HasMultiplyAssign<typename T, typename U> see below;
auto concept HasDivideAssign<typename T, typename U> see below;
auto concept HasModulusAssign<typename T, typename U> see below;
auto concept HasBitAndAssign<typename T, typename U> see below;
auto concept HasBitOrAssign<typename T, typename U> see below;
auto concept HasBitXorAssign<typename T, typename U> see below;
auto concept HasLeftShiftAssign<typename T, typename U> see below;
auto concept HasRightShiftAssign<typename T, typename U> see below;
auto concept HasPreincrement<typename T> see below;
auto concept HasPostincrement<typename T> see below;
auto concept HasPredecrement<typename T> see below;
auto concept HasPostdecrement<typename T> see below;
auto concept HasComma<typename T, typename U> see below;

// 20.1.4, predicates:
auto concept Predicate<typename F, typename... Args> see below;

// 20.1.5, comparisons:
auto concept LessThanComparable<typename T> see below;
auto concept EqualityComparable<typename T> see below;
auto concept StrictWeakOrder<typename F, typename T> see below;
auto concept EquivalenceRelation<typename F, typename T> see below;

// 20.1.6, construction:
auto concept HasConstructor<typename T, typename... Args> see below;
auto concept Constructible<typename T, typename... Args> see below;
auto concept DefaultConstructible<typename T> see below;
```

```
  concept TriviallyDefaultConstructible<typename T> see below;

  // 20.1.7, destruction:
  auto concept HasDestructor<typename T> see below;
  auto concept HasVirtualDestructor<typename T> see below;
  auto concept NothrowDestructible<typename T> see below;
  concept TriviallyDestructible<typename T> see below;

  // 20.1.8, copy and move:
  auto concept MoveConstructible<typename T> see below;
  auto concept CopyConstructible<typename T> see below;
  concept TriviallyCopyConstructible<typename T> see below;
  auto concept MoveAssignable<typename T> see below;
  auto concept CopyAssignable<typename T> see below;
  concept TriviallyCopyAssignable<typename T> see below;
  auto concept HasSwap<typename T, typename U> see below;
  auto concept Swappable<typename T> see below;

  // 20.1.9, memory allocation:
  auto concept FreeStoreAllocatable<typename T> see below;

  // 20.1.10, regular types:
  auto concept Semiregular<typename T> see below;
  auto concept Regular<typename T> see below;

  // 20.1.11, convertibility:
  auto concept ExplicitlyConvertible<typename T, typename U> see below;
  auto concept Convertible<typename T, typename U> see below;


  // 20.1.12, arithmetic concepts:
  concept ArithmeticLike<typename T> see below;
  concept IntegralLike<typename T> see below;
  concept SignedIntegralLike<typename T> see below;
  concept UnsignedIntegralLike<typename T> see below;
  concept FloatingPointLike<typename T> see below;
}
```

20.1.1 Type transformations [concept.transform]

1 The concepts in 20.1.1 provide simple type transformations that can be used within constrained templates.

2 A program shall not provide concept maps for any concept in 20.1.1.

```
auto concept IdentityOf<typename T> {
  typename type = T;
  requires SameType<type, T>;
}
```

3 *Note:* concept form of the `identity` type metafunction (20.6.6).

```
auto concept RvalueOf<typename T> {
  typename type = T&&;
  requires Convertible<T&, type> && Convertible<T&&, type>;
}
```

4 *Note:* describes the rvalue reference type for an arbitrary type `T`.

```
template<typename T> concept_map RvalueOf<T&> {
  typedef T&& type;
}
```

Note: provides the appropriate rvalue reference type for the rvalue an lvalue reference type. [*Note:* this concept map is required to circumvent reference collapsing for lvalue references. — *end note*]

20.1.2 True [concept.true]

```
concept True<bool> { }
concept_map True<true> { }
```

Note: used to express the requirement that a particular integral constant expression evaluate true.

Requires: a program shall not provide a concept map for the `True` concept.

20.1.3 Operator concepts [concept.operator]

```
auto concept HasPlus<typename T, typename U> {
  typename result_type;
  result_type operator+(const T&, const U&);
}
```

Note: describes types with a binary `operator+`.

```
auto concept HasMinus<typename T, typename U> {
  typename result_type;
  result_type operator-(const T&, const U&);
}
```

Note: describes types with a binary `operator-`.

```
auto concept HasMultiply<typename T, typename U> {
  typename result_type;
  result_type operator*(const T&, const U&);
}
```

Note: describes types with a binary `operator*`.

```
auto concept HasDivide<typename T, typename U> {
  typename result_type;
  result_type operator/(const T&, const U&);
}
```

Note: describes types with an `operator/`.

```
auto concept HasModulus<typename T, typename U> {
  typename result_type;
  result_type operator%(const T&, const U&);
}
```

Note: describes types with an `operator%`.

```
auto concept HasUnaryPlus<typename T> {
  typename result_type;
  result_type operator+(const T&);
}
```

6 *Note:* describes types with a unary `operator+`.

```
auto concept HasNegate<typename T> {
  typename result_type;
  result_type operator-(const T&);
}
```

7 *Note:* describes types with a unary `operator-`.

```
auto concept HasLess<typename T, typename U> {
  bool operator<(const T& a, const U& b);
}
```

8 *Note:* describes types with an `operator<`.

```
auto concept HasGreater<typename T, typename U> {
  bool operator>(const T& a, const U& b);
}
```

9 *Note:* describes types with an `operator>`.

```
auto concept HasLessEqual<typename T, typename U> {
  bool operator<=(const T& a, const U& b);
}
```

10 *Note:* describes types with an `operator<=`.

```
auto concept HasGreaterEqual<typename T, typename U> {
  bool operator>=(const T& a, const U& b);
}
```

11 *Note:* describes types with an `operator>=`.

12 For the concepts `HasLess`, `HasGreater`, `HasLessEqual`, and `HasGreaterEqual`, the concept maps in namespace `std` for any pointer type yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not.

```
auto concept HasEqualTo<typename T, typename U> {
  bool operator==(const T& a, const U& b);
}
```

13 *Note:* describes types with an `operator==`.

```
auto concept HasNotEqualTo<typename T, typename U> {
  bool operator!=(const T& a, const U& b);
}
```

14 *Note:* describes types with an `operator!=`.

```
auto concept HasLogicalAnd<typename T, typename U> {
  bool operator&&(const T&, const U&);
}
```

15 *Note:* describes types with a logical conjunction operator.

```
auto concept HasLogicalOr<typename T, typename U> {
  bool operator||(const T&, const U&);
}
```

16 *Note:* describes types with a logical disjunction operator.

```
auto concept HasLogicalNot<typename T> {
```

```
  bool operator!(const T&);
}
```

17 *Note:* describes types with a logical negation operator.

```
auto concept HasBitAnd<typename T, typename U> {
  typename result_type;
  result_type operator&(const T&, const U&);
}
```

18 *Note:* describes types with a binary `operator&`.

```
auto concept HasBitOr<typename T, typename U> {
  typename result_type;
  result_type operator|(const T&, const U&);
}
```

19 *Note:* describes types with an `operator|`.

```
auto concept HasBitXor<typename T, typename U> {
  typename result_type;
  result_type operator^(const T&, const U&);
}
```

20 *Note:* describes types with an `operator^`.

```
auto concept HasComplement<typename T> {
  typename result_type;
  result_type operator~(const T&);
}
```

21 *Note:* describes types with an `operator~`.

```
auto concept HasLeftShift<typename T, typename U> {
  typename result_type;
  result_type operator<<(const T&, const U&);
}
```

22 *Note:* describes types with an `operator`<<.

```
auto concept HasRightShift<typename T, typename U> {
  typename result_type;
  result_type operator>>(const T&, const U&);
}
```

23 *Note:* describes types with an `operator`>>.

```
auto concept HasDereference<typename T> {
  typename result_type;
  result_type operator*(T&&);
}
```

24 *Note:* describes types with a dereferencing `operator*`.

```
auto concept HasAddressOf<typename T> {
  typename result_type;
  result_type operator&(T&);
}
```

25 *Note:* describes types with an address-of `operator&`.

```
auto concept HasSubscript<typename T, typename U> {
  typename result_type;
  result_type operator[](T&&, const U&);
}
```

26 *Note:* describes types with a subscript `operator[]`.

```
auto concept Callable<typename F, typename... Args> {
  typename result_type;
  result_type operator()(F&&, Args...);
}
```

27 *Note:* describes function object types callable given arguments of types `Args...`.

```
auto concept HasAssign<typename T, typename U> {
  typename result_type;
  result_type T::operator=(U);
}
```

28 *Note:* describes types with an assignment operator.

```
auto concept HasPlusAssign<typename T, typename U> {
  typename result_type;
  result_type operator+=(T&, U);
}
```

29 *Note:* describes types with an `operator`+ =.

```
auto concept HasMinusAssign<typename T, typename U> {
  typename result_type;
  result_type operator-=(T&, U);
}
```

30 *Note:* describes types with an `operator`− =.

```
auto concept HasMultiplyAssign<typename T, typename U> {
  typename result_type;
  result_type operator*=(T&, U);
}
```

31 *Note:* describes types with an `operator`∗ =.

```
auto concept HasDivideAssign<typename T, typename U> {
  typename result_type;
  result_type operator/=(T&, U);
}
```

32 *Note:* describes types with an `operator`/ =.

```
auto concept HasModulusAssign<typename T, typename U> {
  typename result_type;
  result_type operator%=(T&, U);
}
```

33 *Note:* describes types with an `operator`% =.

```
auto concept HasBitAndAssign<typename T, typename U> {
  typename result_type;
  result_type operator&=(T&, U);
}
```

34 *Note:* describes types with an `operator&=`.

```
auto concept HasBitOrAssign<typename T, typename U> {
  typename result_type;
  result_type operator|=(T&, U);
}
```

35 *Note:* describes types with an `operator|=`.

```
auto concept HasBitXorAssign<typename T, typename U> {
  typename result_type;
  result_type operator^=(T&, U);
}
```

36 *Note:* describes types with an `operator^=`.

```
auto concept HasLeftShiftAssign<typename T, typename U> {
  typename result_type;
  result_type operator<<=(T&, U);
}
```

37 *Note:* describes types with an `operator`<<=.

```
auto concept HasRightShiftAssign<typename T, typename U> {
  typename result_type;
  result_type operator>>=(T&, U);
}
```

38 *Note:* describes types with an `operator`>>=.

```
auto concept HasPreincrement<typename T> {
  typename result_type;
  result_type operator++(T&);
}
```

39 *Note:* describes types with a pre-increment operator.

```
auto concept HasPostincrement<typename T> {
  typename result_type;
  result_type operator++(T&, int);
}
```

40 *Note:* describes types with a post-increment operator.

```
auto concept HasPredecrement<typename T> {
  typename result_type;
  result_type operator--(T&);
}
```

41 *Note:* describes types with a pre-decrement operator.

```
auto concept HasPostdecrement<typename T> {
  typename result_type;
  result_type operator--(T&, int);
}
```

42 *Note:* describes types with a post-decrement operator.

```
auto concept HasComma<typename T, typename U> {
  typename result_type
```

```
  result_type operator,(const T&, const U&);
}
```

43 *Note:* describes types with a comma operator.

20.1.4 Predicates [concept.predicate]

```
auto concept Predicate<typename F, typename... Args> : Callable<F, const Args&...> {
  requires Convertible<result_type, bool>;
}
```

1 *Note:* describes function objects callable with some set of arguments, the result of which can be used in a context that requires a `bool`.

2 *Requires:* predicate function objects shall not apply any non-constant function through the predicate arguments.

20.1.5 Comparisons [concept.comparison]

```
auto concept LessThanComparable<typename T> : HasLess<T, T> {
  bool operator>(const T& a, const T& b) { return b < a; }
  bool operator<=(const T& a, const T& b) { return !(b < a); }
  bool operator>=(const T& a, const T& b) { return !(a < b); }

  axiom Consistency(T a, T b) {
    (a > b) == (b < a);
    (a <= b) == !(b < a);
    (a >= b) == !(a < b);
  }

  axiom Irreflexivity(T a) { (a < a) == false; }

  axiom Antisymmetry(T a, T b) {
    if (a < b)
      (b < a) == false;
  }

  axiom Transitivity(T a, T b, T c) {
    if (a < b && b < c)
      (a < c) == true;
  }

  axiom TransitivityOfEquivalence(T a, T b, T c) {
    if (!(a < b) && !(b < a) && !(b < c) && !(c < b))
      (!(a < c) && !(c < a)) == true;
  }
}
```

1 *Note:* describes types whose values can be ordered, where `operator<` is a strict weak ordering relation (25.3).

```
auto concept EqualityComparable<typename T> : HasEqualTo<T, T> {
  bool operator!=(const T& a, const T& b) { return !(a == b); }

  axiom Consistency(T a, T b) {
```

```
    (a == b) == !(a != b);
  }

  axiom Reflexivity(T a) { a == a; }

  axiom Symmetry(T a, T b) {
    if (a == b)
      b == a;
  }

  axiom Transitivity(T a, T b, T c) {
    if (a == b && b == c)
      a == c;
  }
}
```

2 *Note:* describes types whose values can be compared for equality with `operator==`, which is an equivalence relation.

```
auto concept StrictWeakOrder<typename F, typename T> : Predicate<F, T, T> {

  axiom Irreflexivity(F f, T a) { f(a, a) == false; }

  axiom Antisymmetry(F f, T a, T b) {
    if (f(a, b))
      f(b, a) == false;
  }

  axiom Transitivity(F f, T a, T b, T c) {
    if (f(a, b) && f(b, c))
      f(a, c) == true;
  }

  axiom TransitivityOfEquivalence(F f, T a, T b, T c) {
    if (!f(a, b) && !f(b, a) && !f(b, c) && !f(c, b))
      (!f(a, c) && !f(c, a)) == true;
  }
}
```

3 *Note:* describes a strict weak ordering relation (25.3), `F`, on a type `T`.

```
auto concept EquivalenceRelation<typename F, typename T> : Predicate<F, T, T> {
  axiom Reflexivity(F f, T a) { f(a, a) == true; }

  axiom Symmetry(F f, T a, T b) {
    if (f(a, b))
      f(b, a) == true;
  }

  axiom Transitivity(F f, T a, T b, T c) {
    if (f(a, b) && f(b, c))
      f(a, c) == true;
  }
}
```

4 *Note:* describes an equivalence relation, `F`, on a type `T`.

20.1.6 Construction [concept.construct]

```
auto concept HasConstructor<typename T, typename... Args> {
  T::T(Args...);
}
```

1 *Note:* describes types that can be constructed from a given set of arguments.

```
auto concept Constructible<typename T, typename... Args>
  : HasConstructor<T, Args...>, NothrowDestructible<T> { }
```

2 *Note:* describes types that can be constructed from a given set of arguments that also have a no-throw destructor.

```
auto concept DefaultConstructible<typename T> : Constructible<T> { }
```

3 *Note:* describes types for which an object can be constructed without initializing the object to any particular value.

```
concept TriviallyDefaultConstructible<typename T> : DefaultConstructible<T> { }
```

4 *Note:* describes types whose default constructor is trivial.

5 *Requires:* for every type `T` that is a trivial type (3.9) or a class type with a trivial default constructor (12.1), a concept map `TriviallyDefaultConstructible<T>` shall be implicitly defined in namespace `std`.

20.1.7 Destruction [concept.destruct]

```
auto concept HasDestructor<typename T> {
  T::~T();
}
```

1 *Note:* describes types that can be destroyed. These are scalar types, references, and class types with a public non-deleted destructor.

```
concept HasVirtualDestructor<typename T> : HasDestructor<T>, PolymorphicClass<T> { }
```

2 *Note:* describes types with a virtual destructor.

3 *Requires:* for every class type `T` that has a virtual destructor, a concept map `HasVirtualDestructor<T>` shall be implicitly defined in namespace `std`.

```
auto concept NothrowDestructible<typename T> : HasDestructor<T> { }
```

`T::~T()` *// inherited from HasDestructor<T>*

4 *Requires:* no exception is propagated.

```
concept TriviallyDestructible<typename T> : NothrowDestructible<T> { }
```

5 *Note:* describes types whose destructors do not need to be executed when the object is destroyed.

6 *Requires:* for every type `T` that is a trivial type (3.9), reference, or class type with a trivial destructor (12.4), a concept map `TriviallyDestructible<T>` shall be implicitly defined in namespace `std`.

20.1.8 Copy and move [concept.copymove]

```
auto concept MoveConstructible<typename T> : Constructible<T, T&&> {
  requires RvalueOf<T> && Constructible<T, RvalueOf<T>::type>;
}
```

1 *Note:* describes types that can move-construct an object from a value of the same type, possibly altering that value.

`T::T(T&& rv);` *// note: inherited from HasConstructor<T, T&&>*

2 *Postcondition:* the constructed `T` object is equivalent to the value of `rv` before the construction. [*Note:* there is no requirement on the value of `rv` after the construction. — *end note*]

```
auto concept CopyConstructible<typename T> : MoveConstructible<T>, Constructible<T, const T&> {
  axiom CopyPreservation(T x) {
    T(x) == x;
  }
}
```

3 *Note:* describes types with a public copy constructor.

```
concept TriviallyCopyConstructible<typename T> : CopyConstructible<T> { }
```

4 *Note:* describes types whose copy constructor is equivalent to `memcpy`.

5 *Requires:* for every type `T` that is a trivial type (3.9), a reference, or a class type with a trivial copy constructor (12.8), a concept map `TriviallyCopyConstructible<T>` shall be implicitly defined in namespace `std`.

```
auto concept MoveAssignable<typename T> : HasAssign<T, T&&> {
  requires RvalueOf<T> && HasAssign<T, RvalueOf<T>::type>;
}
```

6 *Note:* describes types with the ability to assign to an object from an rvalue, potentially altering the rvalue.

`result_type T::operator=(T&& rv);` *// inherited from HasAssign<T, T&&>*

7 *Postconditions:* the constructed `T` object is equivalent to the value of `rv` before the assignment. [*Note:* there is no requirement on the value of `rv` after the assignment. — *end note*]

```
auto concept CopyAssignable<typename T> : HasAssign<T, const T&>, MoveAssignable<T> {
  axiom CopyPreservation(T& x, T y) {
    (x = y, x) == y;
  }
}
```

8 *Note:* describes types with the ability to assign to an object.

```
concept TriviallyCopyAssignable<typename T> : CopyAssignable<T> { }
```

9 *Note:* describes types whose copy-assignment operator is equivalent to `memcpy`.

10 *Requires:* for every type `T` that is a trivial type (ref) or a class type with a trivial copy assignment operator (12.8), a concept map `TriviallyCopyAssignable<T>` shall be implicitly defined in namespace `std`.

```
auto concept HasSwap<typename T, typename U> {
  void swap(T, U);
}
```

11 *Note:* describes types that have a swap operation.

```
auto concept Swappable<typename T> : HasSwap<T&, T&> { }
```

12 *Note:* describes types for which two values of that type can be swapped.

```
void swap(T& t, T& u); // inherited from HasSwap<T, T>
```

13 *Postconditions:* `t` has the value originally held by `u`, and `u` has the value originally held by `t`.

20.1.9 Memory allocation [concept.memory]

```
auto concept FreeStoreAllocatable<typename T> {
  void* T::operator new(size_t size);
  void T::operator delete(void*);

  void* T::operator new[](size_t size) {
    return T::operator new(size);
  }

  void T::operator delete[](void* ptr) {
    T::operator delete(ptr);
  }

  void* T::operator new(size_t size, const nothrow_t&) {
    try {
      return T::operator new(size);
    } catch(...) {
      return 0;
    }
  }

  void* T::operator new[](size_t size, const nothrow_t&) {
    try {
      return T::operator new[](size);
    } catch(...) {
      return 0;
    }
  }

  void T::operator delete(void* ptr, const nothrow_t&) {
    T::operator delete(ptr);
  }

  void T::operator delete[](void* ptr, const nothrow_t&) {
    T::operator delete[](ptr);
  }
}
```

1 *Note:* describes types for which objects and arrays of objects can be allocated on or freed from the free store with `new` and `delete`.

20.1.10 Regular types [concept.regular]

```
auto concept Semiregular<typename T>
```

```
  : CopyConstructible<T>, CopyAssignable<T>, FreeStoreAllocatable<T> {
  requires SameType<CopyAssignable<T>::result_type, T&>;
}
```

1 *Note:* collects several common requirements supported by most types.

```
auto concept Regular<typename T>
  : Semiregular<T>, DefaultConstructible<T>, EqualityComparable<T> { }
```

2 *Note:* describes semi-regular types that are default constructible and have equality comparison operators.

20.1.11 Convertibility [concept.convertible]

```
auto concept ExplicitlyConvertible<typename T, typename U> {
  explicit operator U(const T&);
}
```

1 *Note:* describes types with a conversion (explicit or implicit) from `T` to `U`.

```
auto concept Convertible<typename T, typename U> : ExplicitlyConvertible<T, U> {
  operator U(const T&);
}
```

2 *Note:* describes types with an implicit conversion from `T` to `U`.

20.1.12 Arithmetic concepts [concept.arithmetic]

```
concept ArithmeticLike<typename T>
  : Regular<T>, LessThanComparable<T>, HasUnaryPlus<T>, HasNegate<T>,
    HasPlus<T, T>, HasMinus<T, T>, HasMultiply<T, T>, HasDivide<T, T>,
    HasPreincrement<T>, HasPostincrement<T>, HasPredecrement<T>, HasPostdecrement<T>,
    HasPlusAssign<T, const T&>, HasMinusAssign<T, const T&>,
    HasMultiplyAssign<T, const T&>, HasDivideAssign<T, const T&> {
  explicit T::T(intmax_t);
  explicit T::T(uintmax_t);
  explicit T::T(long double);

  requires Convertible<HasUnaryPlus<T>::result_type, T>
        && Convertible<HasNegate<T>::result_type, T>
        && Convertible<HasPlus<T, T>::result_type, T>
        && Convertible<HasMinus<T, T>::result_type, T>
        && Convertible<HasMultiply<T, T>::result_type, T>
        && Convertible<HasDivide<T, T>::result_type, T>
        && SameType<HasPreincrement<T>::result_type, T&>
        && SameType<HasPostincrement<T>::result_type, T>
        && SameType<HasPredecrement<T>::result_type, T&>
        && SameType<HasPostdecrement<T>::result_type, T>
        && SameType<HasPlusAssign<T, const T&>::result_type, T&>
        && SameType<HasMinusAssign<T, const T&>::result_type, T&>
        && SameType<HasMultiplyAssign<T, const T&>::result_type, T&>
        && SameType<HasDivideAssign<T, const T&>::result_type, T&>;
}
```

1 *Note:* describes types that provide all of the operations available on arithmetic types (3.9.1).

```
concept IntegralLike<typename T>
  : ArithmeticLike<T>,
    HasComplement<T>, HasModulus<T, T>, HasBitAnd<T, T>, HasBitXor<T, T>, HasBitOr<T, T>,
    HasLeftShift<T, T>, HasRightShift<T, T>
    HasModulusAssign<T, const T&>, HasLeftShiftAssign<T, const T&>, HasRightShiftAssign<T, const T&>
    HasBitAndAssign<T, const T&>, HasBitXorAssign<T, const T&>, HasBitOrAssign<T, const T&> {
  requires Convertible<HasComplement<T>::result_type, T>
        && Convertible<HasModulus<T, T>::result_type, T>
        && Convertible<HasBitAnd<T, T>::result_type, T>
        && Convertible<HasBitXor<T, T>::result_type, T>
        && Convertible<HasBitOr<T, T>::result_type, T>
        && Convertible<HasLeftShift<T, T>::result_type, T>
        && Convertible<HasRightShift<T, T>::result_type, T>
        && SameType<HasModulusAssign<T, const T&>::result_type, T&>
        && SameType<HasLeftShiftAssign<T, const T&>::result_type, T&>
        && SameType<HasRightShiftAssign<T, const T&>::result_type, T&>
        && SameType<HasBitAndAssign<T, const T&>::result_type, T&>
        && SameType<HasBitXorAssign<T, const T&>::result_type, T&>
        && SameType<HasBitOrAssign<T, const T&>::result_type, T&>;
}
```

2 *Note:* describes types that provide all of the operations available on integral types.

```
concept SignedIntegralLike<typename T> : IntegralLike<T> { }
```

3 *Note:* describes types that provide all of the operations available on signed integral types.

4 *Requires:* for every signed integral type `T` (3.9.1), including signed extended integral types, an empty concept map `SignedIntegralLike<T>` shall be defined in namespace `std`.

```
concept UnsignedIntegralLike<typename T> : IntegralLike<T> { }
```

5 *Note:* describes types that provide all of the operations available on unsigned integral types.

6 *Requires:* for every unsigned integral type `T` (3.9.1), including unsigned extended integral types, an empty concept map `UnsignedIntegralLike<T>` shall be defined in namespace `std`.

```
concept FloatingPointLike<typename T> : ArithmeticLike<T> { }
```

7 *Note:* describes floating-point types.

8 *Requires:* for every floating point type `T` (3.9.1), an empty concept map `FloatingPointLike<T>` shall be defined in namespace `std`.

20.2 Utility components [utility]

1 This subclause contains some basic function and class templates that are used throughout the rest of the library.

Header <utility> synopsis

```
namespace std {
  // 20.2.1, operators:
  namespace rel_ops {
    template<EqualityComparable T> bool operator!=(const T&, const T&);
    template<LessThanComparable T> bool operator> (const T&, const T&);
    template<LessThanComparable T> bool operator<=(const T&, const T&);
    template<LessThanComparable T> bool operator>=(const T&, const T&);
```

```
  }

  // 20.2.2, forward/move:
  template <IdentityOf T> T&& forward(IdentityOf<T>::type&&);
  template <RvalueOf T> RvalueOf<T>::type move(T&&);

  // 20.2.3, pairs:
  template <VariableType T1,VariableType T2> struct pair;
  template <EqualityComparable T1, EqualityComparable T2>
    bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
  template <LessThanComparable T1, LessThanComparable T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
  template <EqualityComparable T1, EqualityComparable T2>
    bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
  template <LessThanComparable T1, LessThanComparable T2>
    bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
  template <LessThanComparable T1, LessThanComparable T2>
    bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
  template <LessThanComparable T1, LessThanComparable T2>
    bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
  template <Swappable T1, Swappable T2>
    void swap(pair<T1,T2>&, pair<T1,T2>&);
  template <Swappable T1, Swappable T2>
    void swap(pair<T1,T2>&&, pair<T1,T2>&);
  template <Swappable T1, Swappable T2>
    void swap(pair<T1,T2>&, pair<T1,T2>&&);
  template <MoveConstructible T1, MoveConstructible T2>
    pair<V1, V2> make_pair(T1&&, T2&&);

  // 20.2.4, tuple-like access to pair:
  template <IdentityOf T> class tuple_size;
  template <size_t I, IdentityOf T> class tuple_element;

  template <VariableType T1, VariableType T2> struct tuple_size<std::pair<T1, T2> >;
  template <VariableType T1, VariableType T2> struct tuple_element<0, std::pair<T1, T2> >;
  template <VariableType T1, VariableType T2> struct tuple_element<1, std::pair<T1, T2> >;

  template<size_t I, class T1, class T2>
    requires True<(I < 2)>
    P& get(std::pair<T1, T2>&);
  template<size_t I, class T1, class T2>
    requires True<(I < 2)>
    const P& get(const std::pair<T1, T2>&);

  // 20.2.5, range concept maps for pair:
  template<InputIterator Iter>
    concept_map Range<pair<Iter, Iter> > see below;
  template<InputIterator Iter>
    concept_map Range<const pair<Iter, Iter> > see below;
}
```

20.2.1 Operators [operators]

1 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

```
template <EqualityComparable T> bool operator!=(const T& x, const T& y);
```

2 *Returns:* `!(x == y)`.

```
template <LessThanComparable T> bool operator>(const T& x, const T& y);
```

3 *Returns:* `y < x`.

```
template <LessThanComparable T> bool operator<=(const T& x, const T& y);
```

4 *Returns:* `!(y < x)`.

```
template <LessThanComparable T> bool operator>=(const T& x, const T& y);
```

5 *Returns:* `!(x < y)`.

6 In this library, whenever a declaration is provided for an `operator!=`, `operator>`, `operator>=`, or `operator<=`, and requirements and semantics are not explicitly provided, the requirements and semantics are as specified in this clause.

20.2.2 forward/move helpers [forward]

1 The library provides templated helper functions to simplify applying move semantics to an lvalue and to simplify the implementation of forwarding functions.

```
template <IdentityOf T> T&& forward(IdentityOf<T>::type&& t);
```

2 [*Note:* The use of `IdentityOf` in `forward` forces users to explicitly specify the template parameter. This is necessary to get the correct forwarding semantics. — *end note*]

3 *Returns:* `t`.

4 [*Example:*

```
template <class T, class A1, class A2>
shared_ptr<T> factory(A1&& a1, A2&& a2) {
  return shared_ptr<T>(new T(std::forward<A1>(a1), std::forward<A2>(a2)));
}

struct A {
  A(int&, const double&);
};

void g() {
  shared_ptr<A> sp1 = factory<A>(2, 1.414); // error: 2 will not bind to int&
  int i = 2;
  shared_ptr<A> sp2 = factory<A>(i, 1.414); // OK
}
```

5 In the first call to `factory`, `A1` is deduced as `int`, so 2 is forwarded to `A`'s constructor as. In the second call to `factory`, `A1` is deduced as `int&`, so `i` is forwarded to `A`'s constructor as. In both cases, `A2` is deduced as `double`, so 1.414 is forwarded to `A`'s constructor as an rvalue.

— *end example*]

```
template <RvalueOf T> RvalueOf<T>::type move(T&& t);
```

6 *Returns:* `t`.

20.2.3 Pairs [pairs]

1 The library provides a template for heterogeneous pairs of values. The library also provides a matching function template to simplify their construction and several templates that provide access to `pair` objects as if they were `tuple` objects (see 20.4.2.3 and 20.4.2.4).

```
template <VariableType T1, VariableType T2>
struct pair {
  typedef T1 first_type;
  typedef T2 second_type;

  T1 first;
  T2 second;
  requires DefaultConstructible<T1> && DefaultConstructible<T2> pair();
  requires CopyConstructible<T1> && CopyConstructible<T2> pair(const T1& x, const T2& y);
  template<class U, class V>
    requires Constructible<T1, const U&> && Constructible<T2, const V&>
    pair(const pair<U, V>& p);
  template<class U, class V>
    requires Constructible<T1, RvalueOf<U>::type> && Constructible<T2, RvalueOf<V>::type>
    pair(pair<U, V>&& p);
  template<class U, class... Args>
    requires Constructible<T1, U&&> && Constructible<T2, Args&&...>
    pair(U&& x, Args&&... args);

  // allocator-extended constructors
  template<Allocator Alloc>
    requires ConstructibleWithAllocator<T1, Alloc> && ConstructibleWithAllocator<T2, Alloc>
    pair(allocator_arg_t, const Alloc& a);
  template<class U, class V, Allocator Alloc>
    requires ConstructibleWithAllocator<T1, Alloc, const U&>
          && ConstructibleWithAllocator<T2, Alloc, const V&>
    pair(allocator_arg_t, const Alloc& a, const pair<U, V>& p);
  template<class U, class V, Allocator Alloc>
    requires ConstructibleWithAllocator<T1, Alloc, RvalueOf<U>::type>
          && ConstructibleWithAllocator<T2, Alloc, RvalueOf<V>::type>
    pair(allocator_arg_t, const Alloc& a, pair<U, V>&& p);
  template<class U, class... Args, Allocator Alloc>
    requires ConstructibleWithAllocator<T1, Alloc, U&&>
          && ConstructibleWithAllocator<T2, Alloc, Args&&...>
    pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);

  template<class U , class V>
    requires HasAssign<T1, const U&> && HasAssign<T2, const V&>
    pair& operator=(const pair<U , V>& p);
  requires MoveAssignable<T1> && MoveAssignable<T2> pair& operator=(pair&& p);
  template<class U , class V>
    requires HasAssign<T1, RvalueOf<U>::type> && HasAssign<T2, RvalueOf<V>::type>
    pair& operator=(pair<U , V>&& p);

  requires Swappable<T1> && Swappable<T2> void swap(pair&& p);
};

template <class T1, class T2, class Alloc>
  concept_map UsesAllocator<pair<T1, T2>, Alloc> {
    typedef Alloc allocator_type;
```

```
    }
```

```
requires DefaultConstructible<T1> && DefaultConstructible<T2> pair();
```

2 *Effects:* Initializes its members as if implemented: `pair() : first(), second() {}`

```
requires CopyConstructible<T1> && CopyConstructible<T2> pair(const T1& x, const T2& y);
```

3 *Effects:* The constructor initializes `first` with `x` and `second` with `y`.

```
template<class U, class V>
  requires Constructible<T1, const U&> && Constructible<T2, const V&>
  pair(const pair<U, V> &p);
```

4 *Effects:* Initializes members from the corresponding members of the argument, performing implicit conversions as needed.

```
template<class U, class V>
  requires Constructible<T1, RvalueOf<U>::type> && Constructible<T2, RvalueOf<V>::type>
  pair(pair<U, V>&& p);
```

5 *Effects:* The constructor initializes `first` with `std::move(p.first)` and `second` with `std::move(p.second)`.

```
template<class U, class... Args>
  requires Constructible<T1, U&&> && Constructible<T2, Args&&...>
  pair(U&& x, Args&&... args);
```

6 *Effects:* The constructor initializes `first` with `std::forward<U>(x)` and `second` with `stdforward<Args>(args)...`

```
template<Allocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc> && ConstructibleWithAllocator<T2, Alloc>
  pair(allocator_arg_t, const Alloc& a);
```

7 *Effects:* The members `first` and `second` are each constructed as `ConstructibleWithAllocator` objects with constructor arguments `(allocator_arg_t(), a)`.

```
template<class U, class V, Allocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, const U&>
        && ConstructibleWithAllocator<T2, Alloc, const V&>
  pair(allocator_arg_t, const Alloc& a, const pair<U, V>& p);
```

8 *Effects:* The members `first` and `second` are each constructed as `ConstructibleWithAllocator` objects with constructor arguments `(allocator_arg_t(), a, p.first)` and `(allocator_arg_t(), a, p.second)`, respectively.

```
template<class U, class V, Allocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, RvalueOf<U>
        && ConstructibleWithAllocator<T2, Alloc, RvalueOf<V>::type>
  pair(allocator_arg_t, const Alloc& a, pair<U, V>&& p);
```

9 *Effects:* The members `first` and `second` are each constructed as `ConstructibleWithAllocator` objects with constructor arguments `(allocator_arg_t(), a, std::move(p.first))` and `(allocator_arg_t(), a, std::move(p.second))`, respectively.

```
template<class U, class... Args, Allocator Alloc>
  requires ConstructibleWithAllocator<T1, Alloc, U&&>
        && ConstructibleWithAllocator<T2, Alloc, Args&&...>
```

```
pair(allocator_arg_t, const Alloc& a, U&& x, Args&&... args);
```

10 *Effects:* The members `first` and `second` are each constructed as `ConstructibleWithAllocator` objects with constructor arguments `(allocator_arg_t(), a, std::forward<U>(x))` and `(allocator_arg_t(), a, std::forward<Args>(args)...)`, respectively.

```
template<class U , class V>
  requires HasAssign<T1, const U&> && HasAssign<T2, const V&>
  pair& operator=(const pair<U , V>& p);
```

11 *Effects:* Assigns to `first` with `p.first` and to `second` with `p.second`.

12 *Returns:* `*this`.

```
requires MoveAssignable<T1> && MoveAssignable<T2> pair& operator=(pair&& p);
```

13 *Effects:* Assigns to `first` with `std::move(p.first)` and to `second` with `std::move(p.second)`.

14 *Returns:* `*this`.

```
template<class U, class V>
  requires HasAssign<T1, RvalueOf<U>::type> && HasAssign<T2, RvalueOf<V>::type>
  pair& operator=(pair<U, V>&& p);
```

15 *Effects:* Assigns to `first` with `std::move(p.first)` and to `second` with `std::move(p.second)`.

16 *Returns:* `*this`.

```
requires Swappable<T1> && Swappable<T2> void swap(pair&& p);
```

17 *Effects:* Swaps `first` with `p.first` and `second` with `p.second`.

```
template <EqualityComparable T1, EqualityComparable T2>
  bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

18 *Returns:* `x.first == y.first && x.second == y.second`.

```
template <LessThanComparable T1, LessThanComparable T2>
  bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

19 *Returns:* `x.first < y.first || (!(y.first < x.first) && x.second < y.second)`.

```
template<class T1, class T2>
  requires Swappable<T1> && Swappable<T2>
  void swap(pair<T1, T2>& x, pair<T1, T2>& y);
template<class T1, class T2>
  requires Swappable<T1> && Swappable<T2>
  void swap(pair<T1, T2>&& x, pair<T1, T2>& y);
template<class T1, class T2>
  requires Swappable<T1> && Swappable<T2>
  void swap(pair<T1, T2>& x, pair<T1, T2>&& y);
```

20 *Effects:* `x.swap(y)`

```
template <MoveConstructible T1, MoveConstructible T2>
  pair<V1, V2> make_pair(T1&& x, T2&& y);
```

21 *Returns:*

```
pair<V1, V2>(std::forward<T1>(x), std::forward<T2>(y));
```

where `V1` and `V2` are determined as follows: Let `Ui` be `decay<Ti>::type` for each `Ti`. Then each `Vi` is `X&` if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

22 [*Example:* In place of:

```
return pair<int, double>(5, 3.1415926);    // explicit types
```

a C++ program may contain:

```
return make_pair(5, 3.1415926);            // types are deduced
```

— *end example*]

20.2.4 Tuple-like access to pair [pair.astuple]

`tuple_size<pair<T1, T2> >::value`

1 *Returns:* integral constant expression.

2 *Value:* 2.

`tuple_element<0, pair<T1, T2> >::type`

3 *Value:* the type `T1`.

`tuple_element<1, pair<T1, T2> >::type`

4 *Value:* the type T2.

```
template<int I, class T1, class T2>
  requires True<(I < 2)>
  P& get(pair<T1, T2>&);

template<int I, class T1, class T2>
  requires True<(I < 2)>
  const P& get(const pair<T1, T2>&);
```

5 *Return type:* If `I == 0` then `P` is `T1`, otherwise `P` is `T2`.

6 *Returns:* If `I == 0` returns `p.first`, otherwise returns `p.second`.

20.2.5 Range concept maps for `pair` [pair.concepts]

```
template<InputIterator Iter>
concept_map Range<pair<Iter, Iter> > {
  typedef Iter iterator;
  Iter begin(pair<Iter, Iter>& p) { return p.first; }
  Iter end(pair<Iter, Iter>& p) { return p.second; }
}

template<InputIterator Iter>
concept_map Range<const pair<Iter, Iter> > {
  typedef Iter iterator;
  Iter begin(const pair<Iter, Iter>& p) { return p.first; }
  Iter end(const pair<Iter, Iter>& p) { return p.second; }
}
```

1 *Note:* these concept maps adapt a `pair` object that holds two iterators to the `Range` concept.

20.2.6 Class template `bitset` [template.bitset]

Header `<bitset>` synopsis

```
#include <cstddef>              // for size_t
#include <string>
#include <stdexcept>            // for invalid_argument,
                                // out_of_range, overflow_error
#include <iosfwd>               // for istream, ostream
namespace std {
  template <size_t N> class bitset;

  // 20.2.6.3 bitset operators:
  template <size_t N>
    bitset<N> operator&(const bitset<N>&, const bitset<N>&);
  template <size_t N>
    bitset<N> operator|(const bitset<N>&, const bitset<N>&);
  template <size_t N>
    bitset<N> operator^(const bitset<N>&, const bitset<N>&);
  template <class charT, class traits, size_t N>
    basic_istream<charT, traits>&
    operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
  template <class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
}
```

1 The header `<bitset>` defines a class template and several related functions for representing and manipulating fixed-size sequences of bits.

```
namespace std {
  template<size_t N> class bitset {
  public:
    // bit reference:
    class reference {
      friend class bitset;
      reference();
    public:
     ~reference();
      reference& operator=(bool x);             // for b[i] = x;
      reference& operator=(const reference&);   // for b[i] = b[j];
      bool operator~() const;                   // flips the bit
      operator bool() const;                    // for x = b[i];
      reference& flip();                        // for b[i].flip();
    };

    // 20.2.6.1 constructors:
    constexpr bitset();
    constexpr bitset(unsigned long long val);
    template<class charT, class traits, class Allocator>
      explicit bitset(
        const basic_string<charT,traits,Allocator>& str,
        typename basic_string<charT,traits,Allocator>::size_type pos = 0,
        typename basic_string<charT,traits,Allocator>::size_type n =
          basic_string<charT,traits,Allocator>::npos,
          charT zero = charT('0'), charT one = charT('1'));
```

```
    explicit bitset(const char *str);

    // 20.2.6.2 bitset operations:
    bitset<N>& operator&=(const bitset<N>& rhs);
    bitset<N>& operator|=(const bitset<N>& rhs);
    bitset<N>& operator^=(const bitset<N>& rhs);
    bitset<N>& operator<<=(size_t pos);
    bitset<N>& operator>>=(size_t pos);
    bitset<N>& set();
    bitset<N>& set(size_t pos, bool val = true);
    bitset<N>& reset();
    bitset<N>& reset(size_t pos);
    bitset<N>  operator~() const;
    bitset<N>& flip();
    bitset<N>& flip(size_t pos);

    // element access:
    constexpr bool operator[](size_t pos) const;       // for b[i];
    reference operator[](size_t pos);                  // for b[i];

    unsigned long to_ulong() const;
    unsigned long long to_ullong() const;
    template <class charT, class traits, class Allocator>
      basic_string<charT, traits, Allocator>
      to_string(charT zero = charT('0'), charT one = charT('1')) const;
    template <class charT, class traits>
      basic_string<charT, traits, allocator<charT> > to_string() const;
    template <class charT>
      basic_string<charT, char_traits<charT>, allocator<charT> > to_string() const;
    basic_string<char, char_traits<char>, allocator<char> > to_string() const;
    size_t count() const;
    constexpr size_t size()  const;
    bool operator==(const bitset<N>& rhs) const;
    bool operator!=(const bitset<N>& rhs) const;
    constexpr bool test(size_t pos) const;
    bool all() const;
    bool any() const;
    bool none() const;
    bitset<N> operator<<(size_t pos) const;
    bitset<N> operator>>(size_t pos) const;
  };
}
```

2 The template class `bitset<N>` describes an object that can store a sequence consisting of a fixed number of bits, `N`.

3 Each bit represents either the value zero (reset) or one (set). To *toggle* a bit is to change the value zero to one, or the value one to zero. Each bit has a non-negative position `pos`. When converting between an object of class `bitset<N>` and a value of some integral type, bit position `pos` corresponds to the *bit value* `1 <<pos`. The integral value corresponding to two or more bits is the sum of their bit values.

4 The functions described in this subclause can report three kinds of errors, each associated with a distinct exception:

— an *invalid-argument* error is associated with exceptions of type `invalid_argument` (19.1.3);

— an *out-of-range* error is associated with exceptions of type `out_of_range` (19.1.5);

— an *overflow* error is associated with exceptions of type `overflow_error` (19.1.8).

20.2.6.1 `bitset` constructors [bitset.cons]

```
constexpr bitset();
```

1 *Effects:* Constructs an object of class `bitset<N>`, initializing all bits to zero.

```
constexpr bitset(unsigned long long val);
```

2 *Effects:* Constructs an object of class `bitset<N>`, initializing the first `M` bit positions to the corresponding bit values in `val`. `M` is the smaller of `N` and the number of bits in the value representation (section 3.9) of `unsigned long long`. If `M<N`, the remaining bit positions are initialized to zero.

```
template <class charT, class traits, class Allocator>
explicit
bitset(const basic_string<charT, traits, Allocator>& str,
       typename basic_string<charT, traits, Allocator>::size_type pos = 0,
       typename basic_string<charT, traits, Allocator>::size_type n =
         basic_string<charT, traits, Allocator>::npos,
         charT zero = charT('0'), charT one = charT('1'));
```

3 *Requires:* `pos <= str.size()`.

4 *Throws:* `out_of_range` if `pos > str.size()`.

5 *Effects:* Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`.

The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`. The function uses `traits::eq()` to compare the character values.

Otherwise, the function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`. `M` is the smaller of `N` and `rlen`.

6 An element of the constructed string has value zero if the corresponding character in `str`, beginning at position `pos`, is `0` `zero`. Otherwise, the element has the value 1. Character position `pos + M - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions.

7 If `M < N`, remaining bit positions are initialized to zero.

```
explicit bitset(const char *str);
```

8 *Effects:* Constructs an object of class `bitset<N>` as if by `bitset(string(str))`.

20.2.6.2 `bitset` members [bitset.members]

```
bitset<N>& operator&=(const bitset<N>& rhs);
```

1 *Effects:* Clears each bit in `*this` for which the corresponding bit in `rhs` is clear, and leaves all other bits unchanged.

Returns: `*this`.

```
bitset<N>& operator|=(const bitset<N>& rhs);
```

Effects: Sets each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

Returns: `*this`.

```
bitset<N>& operator^=(const bitset<N>& rhs);
```

Effects: Toggles each bit in `*this` for which the corresponding bit in `rhs` is set, and leaves all other bits unchanged.

Returns: `*this`.

```
bitset<N>& operator<<=(size_t pos);
```

Effects: Replaces each bit at position `I` in `*this` with a value determined as follows:

— If `I < pos`, the new value is zero;

— If `I >= pos`, the new value is the previous value of the bit at position `I - pos`.

Returns: `*this`.

```
bitset<N>& operator>>=(size_t pos);
```

Effects: Replaces each bit at position `I` in `*this` with a value determined as follows:

— If `pos >= N - I`, the new value is zero;

— If `pos < N - I`, the new value is the previous value of the bit at position `I + pos`.

Returns: `*this`.

```
bitset<N>& set();
```

Effects: Sets all bits in `*this`.

Returns: `*this`.

```
bitset<N>& set(size_t pos, bool val = true);
```

Requires: `pos` is valid

Throws: `out_of_range` if `pos` does not correspond to a valid bit position.

Effects: Stores a new value in the bit at position `pos` in `*this`. If `val` is nonzero, the stored value is one, otherwise it is zero.

Returns: `*this`.

```
bitset<N>& reset();
```

Effects: Resets all bits in `*this`.

Returns: `*this`.

```
bitset<N>& reset(size_t pos);
```

Requires: `pos` is valid

Throws: `out_of_range` if `pos` does not correspond to a valid bit position.

Effects: Resets the bit at position `pos` in `*this`.

Returns: `*this`.

```
bitset<N> operator~() const;
```

Effects: Constructs an object `x` of class `bitset<N>` and initializes it with `*this`.

Returns: `x.flip()`.

```
bitset<N>& flip();
```

Effects: Toggles all bits in `*this`.

Returns: `*this`.

```
bitset<N>& flip(size_t pos);
```

Requires: `pos` is valid

Throws: `out_of_range` if `pos` does not correspond to a valid bit position.

Effects: Toggles the bit at position `pos` in `*this`.

Returns: `*this`.

```
unsigned long to_ulong() const;
```

Throws: `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long`.

Returns: `x`.

```
unsigned long long to_ullong() const;
```

Throws: `overflow_error` if the integral value `x` corresponding to the bits in `*this` cannot be represented as type `unsigned long long`.

Returns: `x`.

```
template <class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>
to_string(charT zero = charT('0'), charT one = charT('1')) const;
```

Effects: Constructs a string object of the appropriate type and initializes it to a string of length `N` characters. Each character is determined by the value of its corresponding bit position in `*this`. Character position `N - 1` corresponds to bit position zero. Subsequent decreasing character positions correspond to increasing bit positions. Bit value zero becomes the character `zero`, bit value one becomes the character `one`.

Returns: The created object.

```
template <class charT, class traits>
  basic_string<charT, traits, allocator<charT> > to_string() const;
```

Returns: `to_string<charT, traits, allocator<charT> >()`.

```
template <class charT>
  basic_string<charT, char_traits<charT>, allocator<charT> > to_string() const;
```

Returns: `to_string<charT, char_traits<charT>, allocator<charT> >()`.

```
basic_string<char, char_traits<char>, allocator<char> > to_string() const;
```

39 *Returns:* `to_string<char, char_traits<char>, allocator<char> >()`.

```
size_t count() const;
```

40 *Returns:* A count of the number of bits set in `*this`.

```
constexpr size_t size() const;
```

41 *Returns:* `N`.

```
bool operator==(const bitset<N>& rhs) const;
```

42 *Returns:* A nonzero value if the value of each bit in `*this` equals the value of the corresponding bit in `rhs`.

```
bool operator!=(const bitset<N>& rhs) const;
```

43 *Returns:* A nonzero value if `!(*this == rhs)`.

```
constexpr bool test(size_t pos) const;
```

44 *Requires:* `pos` is valid

45 *Throws:* `out_of_range` if `pos` does not correspond to a valid bit position.

46 *Returns:* `true` if the bit at position `pos` in `*this` has the value one.

```
bool all() const;
```

47 *Returns:* `count() == size()`

```
bool any() const;
```

48 *Returns:* `count() != 0`

```
bool none() const;
```

49 *Returns:* `count() == 0`

```
bitset<N> operator<<(size_t pos) const;
```

50 *Returns:* `bitset<N>(*this) <<= pos`.

```
bitset<N> operator>>(size_t pos) const;
```

51 *Returns:* `bitset<N>(*this) >>= pos`.

```
constexpr bool operator[](size_t pos) const;
```

52 *Requires:* `pos` shall be valid.

53 *Throws:* nothing.

54 *Returns:* `test(pos)`.

```
bitset<N>::reference operator[](size_t pos);
```

55 *Requires:* `pos` shall be valid.

56 *Throws:* nothing.

57 *Returns:* An object of type `bitset<N>::reference` such that `(*this)[pos] == this->test(pos)`, and such that `(*this)[pos] = val` is equivalent to `this->set(pos, val)`.

20.2.6.3 `bitset` operators [bitset.operators]

```
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs);
```

1 *Returns:* `bitset<N>(lhs) &= rhs`.

```
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs);
```

2 *Returns:* `bitset<N>(lhs) |= rhs`.

```
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs);
```

3 *Returns:* `bitset<N>(lhs) ^= rhs`.

```
template <class charT, class traits, size_t N>
  basic_istream<charT, traits>&
  operator>>(basic_istream<charT, traits>& is, bitset<N>& x);
```

4 A formatted input function (27.6.1.2).

5 *Effects:* Extracts up to `N` characters from `is`. Stores these characters in a temporary object `str` of type `basic_string<charT, traits>`, then evaluates the expression `x = bitset<N>(str)`. Characters are extracted and stored until any of the following occurs:

- `N` characters have been extracted and stored;
- end-of-file occurs on the input sequence;
- the next input character is neither `is.widen('0')` nor `is.widen('1')` (in which case the input character is not extracted).

6 If no characters are stored in `str`, calls `is.setstate(ios_base::failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

7 *Returns:* `is`.

```
template <class charT, class traits, size_t N>
  basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>& os, const bitset<N>& x);
```

8 *Returns:*

```
os << x.template to_string<charT,traits,allocator<charT> >(
  use_facet<ctype<charT> >(os.getloc()).widen('0'),
  use_facet<ctype<charT> >(os.getloc()).widen('1'))
```

(see 27.6.2.6).

20.3 Compile-time rational arithmetic [ratio]

1 This subclause describes the ratio library. It provides a class template `ratio` which exactly represents any finite rational number with a numerator and denominator representable by compile-time constants of type `intmax_t`.

2 Throughout this subclause, the template argument types `R1` and `R2` shall be specializations of the `ratio` template. Diagnostic required.

3 **Header `<ratio>` synopsis**

```
namespace std {
  template <intmax_t N, intmax_t D = 1> class ratio;

  // ratio arithmetic
  template <class R1, class R2> struct ratio_add;
  template <class R1, class R2> struct ratio_subtract;
  template <class R1, class R2> struct ratio_multiply;
  template <class R1, class R2> struct ratio_divide;

  // ratio comparison
  template <class R1, class R2> struct ratio_equal;
  template <class R1, class R2> struct ratio_not_equal;
  template <class R1, class R2> struct ratio_less;
  template <class R1, class R2> struct ratio_less_equal;
  template <class R1, class R2> struct ratio_greater;
  template <class R1, class R2> struct ratio_greater_equal;

  // convenience SI typedefs
  typedef ratio<1, 1000000000000000000000000> yocto;  // see 20.3.4
  typedef ratio<1,    1000000000000000000000> zepto;  // see 20.3.4
  typedef ratio<1,       1000000000000000000> atto;
  typedef ratio<1,          1000000000000000> femto;
  typedef ratio<1,             1000000000000> pico;
  typedef ratio<1,                1000000000> nano;
  typedef ratio<1,                   1000000> micro;
  typedef ratio<1,                      1000> milli;
  typedef ratio<1,                       100> centi;
  typedef ratio<1,                        10> deci;
  typedef ratio<                       10, 1> deca;
  typedef ratio<                      100, 1> hecto;
  typedef ratio<                     1000, 1> kilo;
  typedef ratio<                  1000000, 1> mega;
  typedef ratio<               1000000000, 1> giga;
  typedef ratio<            1000000000000, 1> tera;
  typedef ratio<         1000000000000000, 1> peta;
  typedef ratio<      1000000000000000000, 1> exa;
  typedef ratio<   1000000000000000000000, 1> zetta;  // see 20.3.4
  typedef ratio<1000000000000000000000000, 1> yotta;  // see 20.3.4
}
```

20.3.1 Class template `ratio` [ratio.ratio]

```
namespace std {
  template <intmax_t N, intmax_t D = 1>
  class ratio {
  public:
    static const intmax_t num;
    static const intmax_t den;
  };
}
```

1 The template argument `D` shall not be zero, and the absolute values of the template arguments `N` and `D` shall be representable by type `intmax_t`. Diagnostic required. [*Note:* These rules ensure that infinite ratios are avoided and that for any negative input, there exists a representable value of its absolute

value which is positive. In a two's complement representation, this excludes the most negative value. — *end note*]

2 The static data members `num` and `den` shall have the following values, where `gcd` represents the greatest common divisor of the absolute values of `N` and `D`:

— `num` shall have the value `sign(N) * sign(D) * abs(N) / gcd`.

— `den` shall have the value `abs(D) / gcd`.

20.3.2 Arithmetic on `ratio` types [ratio.arithmetic]

```
template <class R1, class R2> struct ratio_add {
  typedef see below} type;
;
```

1 The nested typedef `type` shall be a synonym for `ratio<T1, T2>` where `T1` has the value `R1::num * R2::den + R2::num * R1::den` and `T2` has the value `R1::den * R2::den`.

```
template <class R1, class R2> struct ratio_subtract {
  typedef see below} type;
;
```

2 The nested typedef `type` shall be a synonym for `ratio<T1, T2>` where `T1` has the value `R1::num * R2::den - R2::num * R1::den` and `T2` has the value `R1::den * R2::den`.

```
template <class R1, class R2> struct ratio_multiply {
  typedef see below} type;
;
```

3 The nested typedef `type` shall be a synonym for `ratio<T1, T2>` where `T1` has the value `R1::num * R2::num` and `T2` has the value `R1::den * R2::den`.

```
template <class R1, class R2> struct ratio_divide {
  typedef see below} type;
;
```

4 The nested typedef `type` shall be a synonym for `ratio<T1, T2>` where `T1` has the value `R1::num * R2::den` and `T2` has the value `R1::den * R2::num`.

20.3.3 Comparison of `ratio` types [ratio.comparison]

```
template <class R1, class R2> struct ratio_equal
  : integral_constant<bool, see below> { };
```

1 If `R1::num == R2::num` and `R1::den == R2::den`, `ratio_equal<R1, R2>` shall be derived from `integral_constant<bool, true>`; otherwise it shall be derived from `integral_constant<bool, false>`.

```
template <class R1, class R2> struct ratio_not_equal
  : integral_constant<bool, !ratio_equal<R1, R2>::value> { };

template <class R1, class R2> struct ratio_less
  : integral_constant<bool, see below> { };
```

2 If `R1::num * R2::den < R2::num * R1::den`, `ratio_less<R1, R2>` shall be derived from `integral_constant<bool, true>`; otherwise it shall be derived from `integral_constant<bool,`

false>. Implementations may use other algorithms to compute this relationship to avoid overflow. If overflow occurs, a diagnostic is required.

```
template <class R1, class R2> struct ratio_less_equal
  : integral_constant<bool, !ratio_less<R2, R1>::value> { };

template <class R1, class R2> struct ratio_greater
  : integral_constant<bool, ratio_less<R2, R1>::value> { };

template <class R1, class R2> struct ratio_greater_equal
  : integral_constant<bool, !ratio_less<R1, R2>::value> { };
```

20.3.4 SI types for ratio [ratio.si]

1 For each of the typedefs yocto, zepto, zetta, and yotta, if both of the constants used in its specification are representable by intmax_t, the typedef shall be defined; if either of the constants is not representable by intmax_t, the typedef shall not be defined.

20.4 Tuples [tuple]

20.4.1 In general [tuple.general]

1 20.4 describes the tuple library that provides a tuple type as the class template tuple that can be instantiated with any number of arguments. Each template argument specifies the type of an element in the tuple. Consequently, tuples are heterogeneous, fixed-size collections of values.

2 **Header <tuple> synopsis**

```
namespace std {
  // 20.4.2, class template tuple:
  template <VariableType... Types> class tuple;

  // 20.4.2.2, tuple creation functions:
  const unspecified ignore;

  template <MoveConstructible... Types>
    tuple<VTypes...> make_tuple(Types&&...);

  template<VariableType... Types>
    tuple<Types&...> tie(Types&...);

  template <CopyConstructible... TTypes, CopyConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, const tuple<UTypes...>&);
  template <MoveConstructible... TTypes, CopyConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&&, const tuple<UTypes...>&);
  template <CopyConstructible... TTypes, MoveConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>&, tuple<UTypes...>&&);
  template <MoveConstructible... TTypes, MoveConstructible... UTypes>
    tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&&, tuple<UTypes...>&&);

  // 20.4.2.3, tuple helper classes:
  template <IdentityOf T> class tuple_size;  // undefined
  template <VariableType... Types> class tuple_size<tuple<Types...> >;

  template <size_t I, IdentityOf T> class tuple_element;    // undefined
  template <size_t I, VariableType... Types>
```

```
    requires True<(I < sizeof...(Types))> class tuple_element<I, tuple<Types...> >;

  // 20.4.2.4, element access:
  template <size_t I, VariableType... Types>
    requires True<(I < sizeof...(Types))>
    typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>&);

  template <size_t I, VariableType... Types>
    requires True<(I < sizeof...(Types))>
    typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>&);

  // 20.4.2.5, relational operators:
  template<class... TTypes, class... UTypes>
    requires EqualityComparable<TTypes, UTypes>...
    bool operator==(const tuple<TTypes...>&, const tuple<UTypes...>&);

  template<class... TTypes, class... UTypes>
    requires LessThanComparable<TTypes, UTypes>...
    bool operator<(const tuple<TTypes...>&, const tuple<UTypes...>&);

  template<class... TTypes, class... UTypes>
    requires EqualityComparable<TTypes, UTypes>...
    bool operator!=(const tuple<TTypes...>&, const tuple<UTypes...>&);

  template<class... TTypes, class... UTypes>
    requires LessThanComparable<UTypes, TTypes>...
    bool operator>(const tuple<TTypes...>&, const tuple<UTypes...>&);

  template<class... TTypes, class... UTypes>
    requires LessThanComparable<UTypes, TTypes>...
    bool operator<=(const tuple<TTypes...>&, const tuple<UTypes...>&);

  template<class... TTypes, class... UTypes>
    requires LessThanComparable<TTypes, UTypes>...
    bool operator>=(const tuple<TTypes...>&, const tuple<UTypes...>&);

  // 20.4.2.7, swap:
  template <class... Types>
    void swap(tuple<Types...>& x, tuple<Types...>& y);
  template <class... Types>
    void swap(tuple<Types...>&& x, tuple<Types...>& y);
  template <class... Types>
    void swap(tuple<Types...>& x, tuple<Types...>&& y);

  // 20.4.2.8, range concept maps for tuple:
  template<InputIterator Iter>
    concept_map Range<tuple<Iter, Iter> > see below;
  template<InputIterator Iter>
    concept_map Range<const tuple<Iter, Iter> > see below;
} // namespace std
```

20.4.2 Class template `tuple` [tuple.tuple]

```
template <VariableType... Types>
class tuple
```

```
{
public:
  requires DefaultConstructible<Types>... tuple();
  template <class... UTypes>
    requires Constructible<Types, UTypes&&>...
    explicit tuple(UTypes&&...);

  requires CopyConstructible<Types>... tuple(const tuple&);

  template <class... UTypes>
    requires Constructible<Types, const UTypes&>...
    tuple(const tuple<UTypes...>&);
  template <class... UTypes>
    requires Constructible<Types, RvalueOf<UTypes>::type>...
    tuple(tuple<UTypes...>&&);

  template <class... UTypes>
    requires Constructible<Types, const UTypes&>...
  template <class... UTypes>
    requires Constructible<Types, RvalueOf<UTypes>::type>...

  // allocator-extended constructors
  template <Allocator Alloc>
    requires ConstructibleWithAllocator<Types, Alloc>...
    tuple(allocator_arg_t, const Alloc& a);
  template <Allocator Alloc, class... UTypes>
    requires ConstructibleWithAllocator<Types, Alloc, UTypes&&>...
    explicit tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
  template <Allocator Alloc, class... UTypes>
    requires ConstructibleWithAllocator<Types, Alloc, const UTypes&>...
    tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
  template <Allocator Alloc, class... UTypes>
    requires ConstructibleWithAllocator<Types, Alloc, RvalueOf<UTypes>::type>...
    tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
  template <Allocator Alloc, class... UTypes>
    requires ConstructibleWithAllocator<Types, Alloc, const UTypes&>...
    tuple(allocator_arg_t, const Alloc& a, const pair<UTypes...>&);
  template <Allocator Alloc, class... UTypes>
    requires ConstructibleWithAllocator<Types, Alloc, RvalueOf<UTypes>::type>...
    tuple(allocator_arg_t, const Alloc& a, pair<UTypes...>&&);

  requires CopyAssignable<Types>... tuple& operator=(const tuple&);

  template <class... UTypes>
    requires HasAssign<Types, const UTypes&>...
    tuple& operator=(const tuple<UTypes...>&);
  template <class... UTypes>
    requires HasAssign<Types, RvalueOf<UTypes>::type>...
    tuple& operator=(tuple<UTypes...>&&);

  template <class... UTypes>
    requires HasAssign<Types, const UTypes&>...
  template <class... UTypes>
    requires HasAssign<Types, RvalueOf<UTypes>::type>...
```

```
  void swap(tuple&& rhs);
};

template<class... Types, class Alloc>
  concept_map UsesAllocator<tuple<Types...>, Alloc> {
    typedef Alloc allocator_type;
  }
```

20.4.2.1 Construction [tuple.cnstr]

1 For each `tuple` constructor and assignment operator, an exception is thrown only if the construction of one of the types in `Types` throws an exception.

```
requires DefaultConstructible<Types>... tuple();
```

2 *Effects:* Default initializes each element.

```
template <class... UTypes>
  requires Constructible<Types, UTypes&&>...
  tuple(UTypes&&... u);
```

3 *Effects:* Initializes the elements in the tuple with the corresponding value in `std::forward<UTypes>(u)`.

```
requires CopyConstructible<Types>... tuple(const tuple& u);
```

4 *Effects:* Copy constructs each element of `*this` with the corresponding element of `u`.

```
template <class... UTypes>
  requires Constructible<Types, const UTypes&>...
  tuple(const tuple<UTypes...>& u);
```

5 *Effects:* Constructs each element of `*this` with the corresponding element of `u`.

```
template <class... UTypes>
  requires Constructible<Types, RvalueOf<UTypes>::type>...
  tuple(tuple<UTypes...>&& u);
```

6 *Effects:* Move-constructs each element of `*this` with the corresponding element of `u`.

```
template <class... UTypes>
  requires Constructible<Types, const UTypes&>...
  tuple(const pair<UTypes...>&);
```

7 *Effects:* Constructs the first element with `u.first` and the second element with `u.second`.

```
template <class... UTypes>
  requires Constructible<Types, RvalueOf<UTypes>::type>...
  tuple(pair<UTypes...>&&);
```

8 *Effects:* Constructs the first element with `std::move(u.first)` and the second element with `std::move(u.second)`.

```
requires CopyAssignable<Types>... tuple& operator=(const tuple& u);
```

9 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

10 *Returns:* `*this`

```
requires MoveAssignable<Types>... tuple& operator=(tuple&& u);
```

11 *Effects:* Move-assigns each element of `u` to the corresponding element of `*this`.

12 *Returns:* `*this`.

```
template <class... UTypes>
  requires HasAssign<Types, const UTypes&>...
  tuple& operator=(const tuple<UTypes...>& u);
```

13 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

14 *Returns:* `*this`

```
template <class... UTypes>
  requires HasMoveAssign<Types, RvalueOf<UTypes>::type>...
  tuple& operator=(tuple<UTypes...>&& u);
```

15 *Effects:* Move-assigns each element of `u` to the corresponding element of `*this`.

16 *Returns:* `*this`.

```
template <class... UTypes>
  requires HasAssign<Types, const UTypes&>...
  tuple& operator=(const pair<UTypes...>&);
```

17 *Effects:* Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

18 *Returns:* `*this`

19 [*Note:* There are rare conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type `T` and if the parameter passed to the constructor is not of type `T`, but rather a tuple type that is convertible to `T`. The effect of the converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, it it possible to compare the "nesting depths" of the source and target tuples and decide to select the element-wise constructor if the source nesting depth is smaller than the target nesting-depth. This can be accomplished using an `enable_if` template or other tools for constrained templates. — *end note*]

```
template <class... UTypes>
  requires HasAssign<Types, RvalueOf<UTypes>::type>...
  tuple& operator=(pair<UTypes...>&&);
```

20 *Effects:* Assigns `std::move(u.first)` to the first element of `*this` and `std::move(u.second)` to the second element of `*this`.

21 *Returns:* `*this`.

```
template <Allocator Alloc>
  requires ConstructibleWithAllocator<Types, Alloc>...
  tuple(allocator_arg_t, const Alloc& a);
template <Allocator Alloc, class... UTypes>
  requires ConstructibleWithAllocator<Types, Alloc, UTypes&&>...
  explicit tuple(allocator_arg_t, const Alloc& a, UTypes&&...);
template <Allocator Alloc, class... UTypes>
  requires ConstructibleWithAllocator<Types, Alloc, const UTypes&>...
  tuple(allocator_arg_t, const Alloc& a, const tuple<UTypes...>&);
template <Allocator Alloc, class... UTypes>
  requires ConstructibleWithAllocator<Types, Alloc, RvalueOf<UTypes>::type>...
```

```
  tuple(allocator_arg_t, const Alloc& a, tuple<UTypes...>&&);
template <Allocator Alloc, class... UTypes>
  requires ConstructibleWithAllocator<Types, Alloc, const UTypes&>...
  tuple(allocator_arg_t, const Alloc& a, const pair<UTypes...>&);
template <Allocator Alloc, class... UTypes>
  requires ConstructibleWithAllocator<Types, Alloc, RvalueOf<UTypes>::type>...
  tuple(allocator_arg_t, const Alloc& a, pair<UTypes...>&&);
```

22 *Effects:* Equivalent to the preceding constructors except that the allocator argument is passed conditionally to the constructor of each element. Each member is *allocator constructed* (20.7.2.2) with `a`.

20.4.2.2 Tuple creation functions [tuple.creation]

```
template<MoveConstructible... Types>
  tuple<VTypes...> make_tuple(Types&&... t);
```

1 Let `Ui` be `decay<Ti>::type` for each `Ti` in `Types`. Then each `Vi` in `VTypes` is `X&` if `Ui` equals `reference_wrapper<X>`, otherwise `Vi` is `Ui`.

2 *Returns:* `tuple<VTypes...>(std::forward<Types>(t)...)`.

3 [*Example:*

```
int i; float j;
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type

```
tuple<int, int&, const float&>
```

— *end example*]

```
template<VariableType... Types>
  tuple<Types&...> tie(Types&... t);
```

4 *Returns:* `tuple<Types&>(t...)`. When an argument in `t` is `ignore`, assigning any value to the corresponding tuple element has no effect.

5 [*Example:* `tie` functions allow one to create tuples that unpack tuples into variables. `ignore` can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

— *end example*]

```
template <CopyConstructible... TTypes, CopyConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
template <MoveConstructible... TTypes, CopyConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, const tuple<UTypes...>& u);
template <CopyConstructible... TTypes, MoveConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(const tuple<TTypes...>& t, tuple<UTypes...>&& u);
template <MoveConstructible... TTypes, MoveConstructible... UTypes>
  tuple<TTypes..., UTypes...> tuple_cat(tuple<TTypes...>&& t, tuple<UTypes...>&& u);
```

6 *Returns:* A `tuple` object constructed by copy- or move-constructing its first `sizeof...(TTypes)` elements from the corresponding elements of `t` and copy- or move-constructing its last `sizeof...(UTypes)` elements from the corresponding elements of `u`.

20.4.2.3 Tuple helper classes [tuple.helper]

```
template <class... Types>
class tuple_size<tuple<Types...> >
  : public integral_constant<size_t, sizeof...(Types)> { };

template <size_t I, class... Types>
requires True<(I < sizeof...(Types))>
class tuple_element<I, tuple<Types...> > {
public:
  typedef TI type;
};
```

1 *Type:* `TI` is the type of the `I`th element of `Types`, where indexing is zero-based.

20.4.2.4 Element access [tuple.elem]

```
template <size_t I, VariableType... Types>
  requires True<(I < sizeof...(Types))>
  typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);
```

1 *Returns:* A reference to the `I`th element of `t`, where indexing is zero-based.

```
template <size_t I, VariableType... Types>
  requires True<(I < sizeof...(Types))>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t);
```

2 *Returns:* A const reference to the `I`th element of `t`, where indexing is zero-based.

3 [*Note:* Constness is shallow. If a `T` in `Types` is some reference type `X&`, the return type is `X&`, not `const X&`. However, if the element type is non-reference type `T`, the return type is `const T&`. This is consistent with how constness is defined to work for member variables of reference type. — *end note*]]

4 [*Note:* The reason `get` is a nonmember function is that if this functionality had been provided as a member function, code where the type depended on a template parameter would have required using the `template` keyword. — *end note*]

20.4.2.5 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1 *Returns:* `true` iff `get<i>(t) == get<i>(u)` for all `i`. For any two zero-length tuples `e` and `f`, `e == f` returns `true`.

2 *Effects:* The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator<(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

3 *Returns:* The result of a lexicographical comparison between `t` and `u`. The result is defined as: `(bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && t`$_{\text{tail}}$ `<` `u`$_{\text{tail}}$`)`, where `r`$_{\text{tail}}$ for some tuple `r` is a tuple containing all but the first element of `r`. For any two zero-length tuples `e` and `f`, `e < f` returns `false`.

```
template<class... TTypes, class... UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator!=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

4 *Returns:* `!(t == u)`.

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<UTypes, TTypes>...
  bool operator>(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

5 *Returns:* `u < t`.

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<UTypes, TTypes>...
  bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

6 *Returns:* `!(u < t)`

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

7 *Returns:* `!(t < u)`

8 [*Note:* The above definitions for comparison operators do not require `t`$_{\text{tail}}$ (or `u`$_{\text{tail}}$) to be constructed. It may not even be possible, as `t` and `u` are not required to be copy constructible. Also, all comparison operators are short circuited; they do not perform element accesses beyond what is required to determine the result of the comparison. — *end note*]

20.4.2.6 Tuple swap [tuple.swap]

```
void swap(tuple&& rhs);
```

1 *Requires:* each type in `Types` shall be `Swappable`.

2 *Effects:* calls `swap` for each element in `*this` and its corresponding element in `rhs`.

3 *Throws:* nothing, unless one of the element-wise `swap` calls throws an exception.

20.4.2.7 Tuple specialized algorithms [tuple.special]

```
template <class... Types>
  void swap(tuple<Types...>& x, tuple<Types...>& y);
template <class... Types>
  void swap(tuple<Types...>&& x, tuple<Types...>& y);
template <class... Types>
  void swap(tuple<Types...>& x, tuple<Types...>&& y);
```

1 *Effects:* `x.swap(y)`

20.4.2.8 Range concept maps for `tuple` [tuple.concepts]

```
template<InputIterator Iter>
concept_map Range<tuple<Iter, Iter> > {
  typedef Iter iterator;
  Iter begin(tuple<Iter, Iter>& p) { return std::get<0>(p); }
  Iter end(tuple<Iter, Iter>& p) { return std::get<1>(p); }
}

template<InputIterator Iter>
concept_map Range<const tuple<Iter, Iter> > {
  typedef Iter iterator;
  Iter begin(const tuple<Iter, Iter>& p) { return std::get<0>(p); }
  Iter end(const tuple<Iter, Iter>& p) { return std::get<1>(p); }
}
```

1 *Note:* these concept maps adapt a `tuple` object that holds two iterators to the `Range` concept.

20.5 Metaprogramming and type traits [meta]

1 This subclause describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time. It includes type classification traits, type property inspection traits, and type transformations. The type classification traits describe a complete taxonomy of all possible C++ types, and state where in that taxonomy a given type belongs. The type property inspection traits allow important characteristics of types or of combinations of types to be inspected. The type transformations allow certain properties of types to be manipulated.

20.5.1 Requirements [meta.rqmts]

1 A *UnaryTypeTrait* describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be `DefaultConstructible`, `CopyConstructible`, and publicly derived, directly or indirectly, from a specialization of the template `integral_constant` (20.5.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described.

2 A *BinaryTypeTrait* describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be `DefaultConstructible`, `CopyConstructible`, and publicly derived, directly or indirectly, from an instance of the template `integral_constant` (20.5.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described.

3 A *TransformationTrait* modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a nested type named `type`, which shall be a synonym for the modified type.

20.5.2 Header `<type_traits>` synopsis [meta.type.synop]

```
namespace std {
  // 20.5.3, helper class:
  template <class T, T v> struct integral_constant;
  typedef integral_constant<bool, true>  true_type;
  typedef integral_constant<bool, false> false_type;
```

```
// 20.5.4.1, primary type categories:
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// 20.5.4.2, composite type categories:
template <class T> struct is_reference;
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// 20.5.4.3, type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_default_constructor;
template <class T> struct has_trivial_copy_constructor;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_default_constructor;
template <class T> struct has_nothrow_copy_constructor;
template <class T> struct has_nothrow_assign;
template <class T> struct has_virtual_destructor;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// 20.5.5, type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
template <class From, class To> struct is_convertible;

// 20.5.6.1, const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
```

```
  template <class T> struct remove_cv;
  template <class T> struct add_const;
  template <class T> struct add_volatile;
  template <class T> struct add_cv;

  // 20.5.6.2, reference modifications:
  template <class T> struct remove_reference;
  template <class T> struct add_lvalue_reference;
  template <class T> struct add_rvalue_reference;

  // 20.5.6.3, sign modifications:
  template <class T> struct make_signed;
  template <class T> struct make_unsigned;

  // 20.5.6.4, array modifications:
  template <class T> struct remove_extent;
  template <class T> struct remove_all_extents;

  // 20.5.6.5, pointer modifications:
  template <class T> struct remove_pointer;
  template <class T> struct add_pointer;

  // 20.5.7, other transformations:
  template <std::size_t Len, std::size_t Align> struct aligned_storage;
  template <std::size_t Len, class... Types> struct aligned_union;
  template <class T> struct decay;
  template <bool, class T = void> struct enable_if;
  template <bool, class T, class F> struct conditional;
  template <class... T> struct common_type;
} // namespace std
```

1 The behavior of a program that adds specializations for any of the class templates defined in this subclause is undefined unless otherwise specified.

20.5.3 Helper classes [meta.help]

```
namespace std {
  template <class T, T v>
  struct integral_constant {
    static const T value = v;
    typedef T value_type;
    typedef integral_constant<T,v> type;
  };
  typedef integral_constant<bool, true> true_type;
  typedef integral_constant<bool, false> false_type;
}
```

1 The class template `integral_constant` and its associated typedefs `true_type` and `false_type` are used as base classes to define the interface for various type traits.

20.5.4 Unary Type Traits [meta.unary]

1 This sub-clause contains templates that may be used to query the properties of a type at compile time.

2 Each of these templates shall be a *UnaryTypeTrait* (20.5.1), publicly derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`.

20.5.4.1 Primary Type Categories [meta.unary.cat]

1 The primary type categories correspond to the descriptions given in section 3.9 of the C++ standard.

2 For any given type `T`, the result of applying one of these templates to `T` and to *cv-qualified* `T` shall yield the same result.

3 [*Note:* For any given type `T`, exactly one of the primary type categories has a `value` member that evaluates to `true`. — *end note*]

Table 31 — Primary type category predicates

Template	Condition	Comments
`template <class T> struct is_void;`	`T` is `void`	
`template <class T> struct is_integral;`	`T` is an integral type (3.9.1)	
`template <class T> struct is_floating_point;`	`T` is a floating point type (3.9.1)	
`template <class T> struct is_array;`	`T` is an array type (3.9.2) of known or unknown extent	Class template `array` (23.2.1) is not an array type.
`template <class T> struct is_pointer;`	`T` is a pointer type (3.9.2)	Includes pointers to functions but not pointers to non-static members.
`template <class T> struct is_lvalue_reference;`	`T` is an lvalue reference type (8.3.2)	
`template <class T> struct is_rvalue_reference;`	`T` is an rvalue reference type (8.3.2)	
`template <class T> struct is_member_object_pointer;`	`T` is a pointer to non-static data member	
`template <class T> struct is_member_function_pointer;`	`T` is a pointer to non-static member function	
`template <class T> struct is_enum;`	`T` is an enumeration type (3.9.2)	
`template <class T> struct is_union;`	`T` is a union type (3.9.2)	
`template <class T> struct is_class;`	`T` is a class type but not a union type (3.9.2)	
`template <class T> struct is_function;`	`T` is a function type (3.9.2)	

20.5.4.2 Composite type traits [meta.unary.comp]

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.

2 For any given type T, the result of applying one of these templates to T, and to *cv-qualified* T shall yield the same result.

Table 32 — Composite type category predicates

Template	Condition	Comments
`template <class T> struct is_reference;`	T is an lvalue reference or an rvalue reference	
`template <class T> struct is_arithmetic;`	T is an arithmetic type (3.9.1)	
`template <class T> struct is_fundamental;`	T is a fundamental type (3.9.1)	
`template <class T> struct is_object;`	T is an object type (3.9)	
`template <class T> struct is_scalar;`	T is a scalar type (3.9)	
`template <class T> struct is_compound;`	T is a compound type (3.9.2)	
`template <class T> struct is_member_pointer;`	T is a pointer to non-static data member or non-static member function	

20.5.4.3 Type properties [meta.unary.prop]

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial specialisations of any of these templates.

3 For all of the class templates X declared in this Clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of X require that the argument must be a complete type.

Table 33 — Type property predicates

Template	Condition	Preconditions
`template <class T> struct is_const;`	T is const-qualified (3.9.3)	
`template <class T> struct is_volatile;`	T is volatile-qualified (3.9.3)	
`template <class T> struct is_trivial;`	T is a trivial type (3.9)	T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct is_standard_layout;`	T is a standard-layout type (3.9)	T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct is_pod;`	T is a POD type (3.9)	T shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.

Table 33 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T> struct is_empty;`	`T` is a class type, but not a union type, with no non-static data members other than bit-fields of length 0, no virtual member functions, no virtual base classes, and no base class `B` for which `is_empty<B>::value` is false.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct is_polymorphic;`	`T` is a polymorphic class (10.3)	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct is_abstract;`	`T` is an abstract class (10.4)	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_trivial_default_constructor;`	`T` is a trivial type (3.9) or a class type with a trivial default constructor (12.1) or an array of such a class type.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_trivial_copy_constructor;`	`T` is a trivial type (3.9) or a reference type or a class type whose copy constructors (12.8) are all trivial.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_trivial_assign;`	`T` is neither `const` nor a reference type, and `T` is a trivial type (3.9) or a class type whose copy assignment operators (12.8) are all trivial.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_trivial_destructor;`	`T` is a trivial type (3.9) or a reference type or a class type with a trivial destructor (12.4) or an array of such a class type.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_nothrow_default_constructor;`	`has_trivial_default_constructor<T>::value` is true or `T` is a class type with a default constructor that is known not to throw any exceptions or `T` is an array of such a class type.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.

Table 33 — Type property predicates (continued)

Template	Condition	Preconditions
`template <class T> struct has_nothrow_copy_constructor;`	`has_trivial_copy_constructor<T>::value` is true or `T` is a class type whose copy constructors are all known not to throw any exceptions or `T` is an array of such a class type.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_nothrow_assign;`	`T` is neither `const` nor a reference type, and `has_trivial_assign<T>::value` is true or `T` is a class type whose copy assignment operators taking an lvalue of type `T` are all known not to throw any exceptions or `T` is an array of such a class type.	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct has_virtual_destructor;`	`T` has a virtual destructor (12.4)	`T` shall be a complete type, an array of unknown bound, or (possibly cv-qualified) `void`.
`template <class T> struct is_signed;`	`is_arithmetic<T>::value && T(-1) < T(0)`	
`template <class T> struct is_unsigned;`	`is_arithmetic<T>::value && T(0) < T(-1)`	

4 [*Example:*

```
is_const<const volatile int>::value     // true
is_const<const int*>::value             // false
is_const<const int&>::value             // false
is_const<int[3]>::value                 // false
is_const<const int[3]>::value           // true
```

— *end example*]

5 [*Example:*

```
remove_const<const volatile int>::type  // volatile int
remove_const<const int* const>::type    // const int*
remove_const<const int&>::type          // const int&
remove_const<const int[3]>::type        // int[3]
```

— *end example*]

Table 34 — Type property queries

Template	Value
`template <class T> struct alignment_of;`	`alignof(T)`. *Precondition:* `T` shall be a complete type, a reference type, or an array of unknown bound, but shall not be a function type or (possibly cv-qualified) `void`.
`template <class T> struct rank;`	If `T` names an array type, an integer value representing the number of dimensions of `T`; otherwise, 0.
`template <class T, unsigned I = 0> struct extent;`	If `T` is not an array type (8.3.4), or if it has rank less than `I`, or if `I` is 0 and `T` has type "array of unknown bound of `U`", then 0; otherwise, the size of the `I`'th dimension of `T`

6 [*Example:*

```
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);
```

— *end example*]

7 [*Example:*

```
// the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert(extent<int[][4]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[][4], 1>::value) == 4);
```

— *end example*]

20.5.5 Relationships between types [meta.rel]

1 This sub-clause contains templates that may be used to query relationships between types at compile time.

2 Each of these templates shall be a *BinaryTypeTrait* (20.5.1), publicly derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`.

Table 35 — Type relationship predicates

Template	Condition	Comments
`template <class T, class U> struct is_same;`	`T` and `U` name the same type with the same cv-qualifications	

Table 35 — Type relationship predicates (continued)

Template	Condition	Comments
`template <class Base, class Derived> struct is_base_of;`	`Base` is a base class of `Derived` (10) without regard to cv-qualifiers or `Base` and `Derived` are not unions and name the same class type without regard to cv-qualifiers	If `Base` and `Derived` are class types and are different types (ignoring possible cv-qualifiers) then `Derived` shall be a complete type. [*Note:* Base classes that are private, protected, or ambigious are, nonetheless, base classes. — *end note*]
`template <class From, class To> struct is_convertible;`	The code set out below shall be well formed.	`From` and `To` shall be complete types, arrays of unknown bound, or (possibly cv-qualified) `void` types.

3 [*Example:*

```
struct B {};
struct B1 : B {};
struct B2 : B {};
struct D : private B1, private B2 {};

is_base_of<B, D>::value         // true
is_base_of<const B, D>::value   // true
is_base_of<B, const D>::value   // true
is_base_of<B, const B>::value   // true
is_base_of<D, B>::value         // false
is_base_of<B&, D&>::value       // false
is_base_of<B[3], D[3]>::value   // false
is_base_of<int, int>::value     // false
```

— *end example*]

4 In order to instantiate the template `is_convertible<From, To>`, the following code shall be well formed:

```
template <class T>
typename add_rvalue_reference<T>::type create();
To test() {
  return create<From>();
}
```

[*Note:* This requirement gives well defined results for reference types, void types, array types, and function types. — *end note*]

20.5.6 Transformations between types [meta.trans]

1 This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this subclause shall be a *TransformationTrait* (20.5.1).

20.5.6.1 Const-volatile modifications [meta.trans.cv]

Table 36 — Const-volatile modifications

Template	Comments
`template <class T> struct remove_const;`	The member typedef `type` shall name the same type as `T` except that any top-level const-qualifier has been removed. [*Example:* `remove_const<const volatile int>::type` evaluates to `volatile int`, whereas `remove_const<const int*>` is `const int*`. — *end example*]
`template <class T> struct remove_volatile;`	The member typedef `type` shall name the same type as `T` except that any top-level volatile-qualifier has been removed. [*Example:* `remove_volatile<const volatile int>::type` evaluates to `const int`, whereas `remove_volatile<volatile int*>` is `volatile int*`. — *end example*]
`template <class T> struct remove_cv;`	The member typedef `type` shall be the same as `T` except that any top-level cv-qualifier has been removed. [*Example:* `remove_cv<const volatile int>::type` evaluates to `int`, whereas `remove_cv<const volatile int*>` is `const volatile int*`. — *end example*]
`template <class T> struct add_const;`	If `T` is a reference, function, or top-level const-qualified type, then `type` shall name the same type as `T`, otherwise `T const`.
`template <class T> struct add_volatile;`	If `T` is a reference, function, or top-level volatile-qualified type, then `type` shall name the same type as `T`, otherwise `T volatile`.
`template <class T> struct add_cv;`	The member typedef `type` shall name the same type as `add_const<typename add_volatile<T>::type>::type`.

20.5.6.2 Reference modifications [meta.trans.ref]

Table 37 — Reference modifications

Template	Comments
`template <class T> struct remove_reference;`	If `T` has type "reference to `T1`" then the member typedef `type` shall name `T1`; otherwise, `type` shall name `T`.
`template <class T> struct add_lvalue_reference;`	If `T` names an object or function type then the member typedef `type` shall name `T&`; otherwise, if `T` names a type "rvalue reference to `T1`" then the member typedef `type` shall name `T1&`; otherwise, `type` shall name `T`.
`template <class T> struct add_rvalue_reference`	If `T` names an object or function type then the member typedef `type` shall name `T&&`; otherwise, `type` shall name `T`. [*Note:* This rule reflects the semantics of reference collapsing. For example, when a type `T` names a type `T1&`, the type `add_rvalue_reference<T>::type` is not an rvalue reference. — *end note*]

20.5.6.3 Sign modifications [meta.trans.sign]

Table 38 — Sign modifications

Template	Comments
`template <class T> struct make_signed;`	If `T` names a (possibly cv-qualified) signed integral type (3.9.1) then the member typedef `type` shall name the type `T`; otherwise, if `T` names a (possibly cv-qualified) unsigned integral type then `type` shall name the corresponding signed integral type, with the same cv-qualifiers as `T`; otherwise, `type` shall name the signed integral type with smallest rank (4.13) for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. *Requires:* `T` shall be a (possibly cv-qualified) integral type or enumeration but not a `bool` type.
`template <class T> struct make_unsigned;`	If `T` names a (possibly cv-qualified) unsigned integral type (3.9.1) then the member typedef `type` shall name the type `T`; otherwise, if `T` names a (possibly cv-qualified) signed integral type then `type` shall name the corresponding unsigned integral type, with the same cv-qualifiers as `T`; otherwise, `type` shall name the unsigned integral type with smallest rank (4.13) for which `sizeof(T) == sizeof(type)`, with the same cv-qualifiers as `T`. *Requires:* `T` shall be a (possibly cv-qualified) integral type or enumeration but not a `bool` type.

20.5.6.4 Array modifications [meta.trans.arr]

Table 39 — Array modifications

Template	Comments
`template <class T> struct remove_extent;`	If `T` names a type "array of `U`", the member typedef `type` shall be `U`, otherwise `T`. [*Note:* For multidimensional arrays, only the first array dimension is removed. For a type "array of `const U`", the resulting type is `const U`. — *end note*]
`template <class T> struct remove_all_extents;`	If `T` is "multi-dimensional array of `U`", the resulting member typedef `type` is `U`, otherwise `T`.

1 [*Example*

```
// the following assertions hold:
assert((is_same<remove_extent<int>::type, int>::value));
assert((is_same<remove_extent<int[2]>::type, int>::value));
assert((is_same<remove_extent<int[2][3]>::type, int[3]>::value));
assert((is_same<remove_extent<int[][3]>::type, int[3]>::value));
```

— *end example*]

2 [*Example*

```
// the following assertions hold:
assert((is_same<remove_all_extents<int>::type, int>::value));
assert((is_same<remove_all_extents<int[2]>::type, int>::value));
assert((is_same<remove_all_extents<int[2][3]>::type, int>::value));
assert((is_same<remove_all_extents<int[][3]>::type, int>::value));
```

— *end example*]

20.5.6.5 Pointer modifications [meta.trans.ptr]

Table 40 — Pointer modifications

Template	Comments
`template <class T> struct remove_pointer;`	If `T` has type "(possibly cv-qualified) pointer to `T1`" then the member typedef `type` shall name `T1`; otherwise, it shall name `T`.
`template <class T> struct add_pointer;`	The member typedef `type` shall name the same type as `remove_reference<T>::type*`.

20.5.7 Other transformations [meta.trans.other]

Table 41 — Other transformations

Template	Condition	Comments
`template <std::size_t Len, std::size_t Align = default-alignment> struct aligned_storage;`	`Len` shall not be zero. `Align` shall be equal to `alignment_of<T>::value` for some type `T` or to *default-alignment*.	The value of *default-alignment* shall be the most stringent alignment requirement for any C++ object type whose size is no greater than `Len` (3.9). The member typedef `type` shall be a POD type suitable for use as uninitialized storage for any object whose size is at most *Len* and whose alignment is a divisor of *Align*.
`template <class T> struct decay;`		Let `U` be `remove_reference<T>::type`. If `is_array<U>::value` is `true`, the member typedef `type` shall equal `remove_extent<U>::type*`. If `is_function<U>::value` is `true`, the member typedef `type` shall equal `add_pointer<U>::type`. Otherwise the member typedef `type` equals `remove_cv<U>::type`.
`template <bool B, class T = void> struct enable_if;`		If `B` is `true`, the member typedef `type` shall equal `T`; otherwise, there shall be no member typedef `type`
`template <bool B, class T, class F> struct conditional;`		If `B` is `true`, the member typedef `type` shall equal `T`. If `B` is `false`, the member typedef `type` shall equal `F`.

Table 41 — Other transformations (continued)

Template	Condition	Comments
`template <class... T> struct common_type;`		The member typedef `type` shall be defined as set out below. All types in the parameter pack `T` shall be complete. A program may specialize this trait if at least one template parameter in the specialization is a user-defined type. [*Note:* Such specializations are needed when only explicit conversion are desired among the template arguments. — *end note*]

1 [*Note:* A typical implementation would define `aligned_storage` as:

```
template <std::size_t Len, std::size_t Alignment>
struct aligned_storage {
  typedef struct {
    unsigned char __data [[ align(Alignment) ]] [Len];
  } type;
};
```

— *end note*]

2 It is implementation-defined whether any extended alignment is supported (3.11).

3 The nested typedef `common_type::type` shall be defined as follows:

```
template <class ...T> struct common_type;

template <class T>
struct common_type<T> {
  typedef T type;
};

template <class T, class U>
struct common_type<T, U> {
private:
  static T&& __t();
  static U&& __u();
public:
  typedef decltype(true ? __t() : __u()) type;
};

template <class T, class U, class... V>
struct common_type<T, U, V...> {
  typedef typename common_type<typename common_type<T, U>::type, V...>::type type;
};
```

20.6 Function objects [function.objects]

1 Function objects are objects with an `operator()` defined. In the places where one would expect to pass a pointer to a function to an algorithmic template (clause 25), the interface is specified to accept

an object with an `operator()` defined. This not only makes algorithmic templates work with pointers to functions, but also enables them to work with arbitrary function objects.

2 **Header `<functional>` synopsis**

```
namespace std {
  // 20.6.3, base:
  template <class Arg, class Result> struct unary_function;
  template <class Arg1, class Arg2, class Result> struct binary_function;

  // 20.6.4 result_of:
  template <class> class result_of;    // undefined
  template <class F, class... Args> class result_of<F(ArgTypes...)>;

  // 20.6.5, reference_wrapper:
  template <ObjectType T> class reference_wrapper;

  template <ObjectType T> reference_wrapper<T> ref(T&);
  template <ObjectType T> reference_wrapper<const T> cref(const T&);

  template <ObjectType T> reference_wrapper<T> ref(reference_wrapper<T>);
  template <ObjectType T> reference_wrapper<const T> cref(reference_wrapper<T>);

  // 20.6.6, identity operation:
  template <IdentityOf T> struct identity;

  // 20.6.7, arithmetic operations:
  template <ReferentType T> struct plus;
  template <ReferentType T> struct minus;
  template <ReferentType T> struct multiplies;
  template <ReferentType T> struct divides;
  template <ReferentType T> struct modulus;
  template <ReferentType T> struct negate;

  // 20.6.8, comparisons:
  template <ReferentType T> struct equal_to;
  template <ReferentType T> struct not_equal_to;
  template <ReferentType T> struct greater;
  template <ReferentType T> struct less;
  template <ReferentType T> struct greater_equal;
  template <ReferentType T> struct less_equal;

  // 20.6.9, logical operations:
  template <ReferentType T> struct logical_and;
  template <ReferentType T> struct logical_or;
  template <ReferentType T> struct logical_not;

  // 20.6.10, bitwise operations:
  template <ReferentType T> struct bit_and;
  template <ReferentType T> struct bit_or;
  template <ReferentType T> struct bit_xor;

  // 20.6.11, negators:
  template <class Predicate> class unary_negate;
  template <class Predicate>
    unary_negate<Predicate>  not1(const Predicate&);
```

```
template <class Predicate> class binary_negate;
template <class Predicate>
  binary_negate<Predicate> not2(const Predicate&);

// 20.6.12, bind:
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<CopyConstructible Fn, CopyConstructible... Types>
  unspecified  bind(Fn, Types...);
template<Returnable R, CopyConstructible Fn, CopyConstructible... Types>
  unspecified  bind(Fn, Types...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  extern unspecified  _1;
  extern unspecified  _2;
              .
              .
              .
  extern unspecified  _M;
}

// D.8, binders (deprecated):
template <class Fn> class binder1st;
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn&, const T&);
template <class Fn> class binder2nd;
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn&, const T&);

// 20.6.13, adaptors:
template <CopyConstructible Arg, Returnable Result>
  class pointer_to_unary_function;
template <CopyConstructible Arg, Returnable Result>
  pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template <CopyConstructible Arg1, CopyConstructible Arg2, Returnable Result>
  class pointer_to_binary_function;
template <CopyConstructible Arg1, CopyConstructible Arg2, Returnable Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

// 20.6.14, adaptors:
template<Returnable S, ClassType T> class mem_fun_t;
template<Returnable S, ClassType T, MoveConstructible A> class mem_fun1_t;
template<Returnable S, ClassType T>
    mem_fun_t<S,T> mem_fun(S (T::*f)());
template<Returnable S, ClassType T, MoveConstructible A>
    mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<Returnable S, ClassType T> class mem_fun_ref_t;
template<Returnable S, ClassType T, CopyConstructible A> class mem_fun1_ref_t;
template<Returnable S, ClassType T>
    mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template<Returnable S, ClassType T, CopyConstructible A>
    mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));
```

```
template <Returnable S, ClassType T> class const_mem_fun_t;
template <Returnable S, ClassType T, CopyConstructible A> class const_mem_fun1_t;
template <Returnable S, ClassType T>
  const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
template <Returnable S, ClassType T, CopyConstructible A>
  const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <Returnable S, ClassType T> class const_mem_fun_ref_t;
template <Returnable S, ClassType T, CopyConstructible A> class const_mem_fun1_ref_t;
template <Returnable S, ClassType T>
  const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <Returnable S, ClassType T, CopyConstructible A>
  const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A) const);

// 20.6.15, member function adaptors:
template<Returnable R, class T> unspecified  mem_fn(R T::*);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...));
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...) const);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...) volatile);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...) const volatile);

// 20.6.16 polymorphic function wrappers:
class bad_function_call;

template<FunctionType> class function; // undefined
template<Returnable R, CopyConstructible... ArgTypes>
  class function<R(ArgTypes...)>;

template<Returnable R, CopyConstructible... ArgTypes>
  void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

template<Returnable R, CopyConstructible... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, nullptr_t);
template<Returnable R, CopyConstructible... ArgTypes>
  bool operator==(nullptr_t, const function<R(ArgTypes...)>&);
template<Returnable R, CopyConstructible... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>&, nullptr_t);
template<Returnable R, CopyConstructible... ArgTypes>
  bool operator!=(nullptr_t, const function<R(ArgTypes...)>&);

// 20.6.17, hash function base template:
template <ReferentType T> struct hash;

// Hash function specializations
template <> struct hash<bool>;
template <> struct hash<char>;
template <> struct hash<signed char>;
template <> struct hash<unsigned char>;
template <> struct hash<char16_t>;
template <> struct hash<char32_t>;
template <> struct hash<wchar_t>;
```

```
  template <> struct hash<short>;
  template <> struct hash<unsigned short>;
  template <> struct hash<int>;
  template <> struct hash<unsigned int>;
  template <> struct hash<long>;
  template <> struct hash<long long>;
  template <> struct hash<unsigned long>;
  template <> struct hash<unsigned long long>;

  template <> struct hash<float>;
  template <> struct hash<double>;
  template <> struct hash<long double>;

  template<PointeeType T> struct hash<T*>;

  template <> struct hash<std::string>;
  template <> struct hash<std::u16string>;
  template <> struct hash<std::u32string>;
  template <> struct hash<std::wstring>;

  template <class Allocator> struct hash<std::vector<bool, Allocator> >;
  template <std::size_t N> struct hash<std::bitset<N> >;
}
```

20.6.1 Definitions [func.def]

1 The following definitions apply to this Clause:

2 A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

3 A *callable type* is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.

4 A *callable object* is an object of a callable type.

5 A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.

6 A *call wrapper* is an object of a call wrapper type.

7 A *target object* is the callable object held by a call wrapper.

20.6.2 Requirements [func.require]

1 Define *INVOKE*(f, t1, t2, ..., tN) as follows:

- (t1.*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;
- ((*t1).*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is not one of the types described in the previous item;
- t1.*f when f is a pointer to member data of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;

— `(*t1).*f` when `f` is a pointer to member data of a class `T` and `t1` is not one of the types described in the previous item;

— `f(t1, t2, ..., tN)` in all other cases.

2 Define *INVOKE*`(f, t1, t2, ..., tN, R)` as *INVOKE*`(f, t1, t2, ..., tN)` implicitly converted to `R`.

3 If a call wrapper (20.6.1) has a *weak result type* the type of its member type `result_type` is based on the type `T` of the wrapper's target object (20.6.1):

— if `T` is a function, reference to function, or pointer to function type, `result_type` shall be a synonym for the return type of `T`;

— if `T` is a pointer to member function, `result_type` shall be a synonym for the return type of `T`;

— if `T` is a class type with a member type `result_type`, then `result_type` shall be a synonym for `T::result_type`;

— otherwise `result_type` shall not be defined.

4 Every call wrapper (20.6.1) shall be CopyConstructible. A *simple call wrapper* is a call wrapper that is CopyAssignable and whose copy constructor and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called with an argument list. [*Note:* in a typical implementation forwarding call wrappers have an overloaded function call operator of the form

```
template<class... ArgTypes>
R operator()(ArgTypes&&... args) cv-qual;
```

— *end note*]

20.6.3 Base [base]

1 The following classes are provided to simplify the typedefs of the argument and result types:

```
namespace std {
  template <class Arg, class Result>
  struct unary_function {
    typedef Arg    argument_type;
    typedef Result result_type;
  };
}

namespace std {
  template <class Arg1, class Arg2, class Result>
  struct binary_function {
    typedef Arg1   first_argument_type;
    typedef Arg2   second_argument_type;
    typedef Result result_type;
  };
}
```

20.6.4 Function object return types [func.ret]

```
namespace std {
  template <class> class result_of;  // undefined
```

```
  template <class Fn, class... ArgTypes>
  class result_of<Fn(ArgTypes...)> {
  public :
    // types
    typedef see below type;
  };
}
```

1 Given an rvalue `fn` of type `Fn` and values `t1, t2, ..., tN` of types `T1, T2, ..., TN` in `ArgTypes`, respectively, the `type` member is the result type of the expression `fn(t1, t2, ...,tN)`. The values `ti` are lvalues when the corresponding type `Ti` is a reference type, and rvalues otherwise.

20.6.5 Class template `reference_wrapper` [refwrap]

```
template <ObjectType T> class reference_wrapper
  : public unary_function<T1, R>          // see below
  : public binary_function<T1, T2, R>     // see below
{
public :
  // types
  typedef T type;
  typedef see below result_type; // Not always defined

  // construct/copy/destroy
  explicit reference_wrapper(T&);
  reference_wrapper(const reference_wrapper<T>& x);

  // assignment
  reference_wrapper& operator=(const reference_wrapper<T>& x);

  // access
  operator T& () const;
  T& get() const;

  // invoke
  template <class... ArgTypes>
    requires Callable<T, ArgTypes&&...>
    Callable<T, ArgTypes&&...>::result_type
  operator() (ArgTypes&&...) const;
};
```

1 `reference_wrapper<T>` is a CopyConstructible and Assignable wrapper around a reference to an object of type `T`.

2 `reference_wrapper` has a weak result type (20.6.2).

3 The template instantiation `reference_wrapper<T>` shall be derived from `std::unary_function<T1, R>` only if the type `T` is any of the following:

— a function type or a pointer to function type taking one argument of type `T1` and returning `R`

— a pointer to member function `R T0::f` *cv* (where *cv* represents the member function's cv-qualifiers); the type `T1` is *cv* `T0*`

— a class type that is derived from `std::unary_function<T1, R>`

4 The template instantiation `reference_wrapper<T>` shall be derived from `std::binary_function<T1, T2, R>` only if the type `T` is any of the following:

— a function type or a pointer to function type taking two arguments of types `T1` and `T2` and returning `R`

— a pointer to member function `R T0::f(T2)` *cv* (where *cv* represents the member function's cv-qualifiers); the type `T1` is *cv* `T0*`

— a class type that is derived from `std::binary_function<T1, T2, R>`

20.6.5.1 `reference_wrapper` construct/copy/destroy [refwrap.const]

```
reference_wrapper(T& t);
```

1 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `t`.

2 *Throws:* nothing.

```
reference_wrapper(const reference_wrapper<T>& x);
```

3 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

4 *Throws:* nothing.

20.6.5.2 `reference_wrapper` assignment [refwrap.assign]

```
reference_wrapper& operator=(const reference_wrapper<T>& x);
```

1 *Postconditions:* `*this` stores a reference to `x.get()`.

2 *Throws:* nothing.

20.6.5.3 `reference_wrapper` access [refwrap.access]

```
operator T& () const;
```

1 *Returns:* The stored reference.

2 *Throws:* nothing.

```
T& get() const;
```

3 *Returns:* The stored reference.

4 *Throws:* nothing.

20.6.5.4 `reference_wrapper` invocation [refwrap.invoke]

```
template <class... ArgTypes>
  requires Callable<T, ArgTypes&&...>
  Callable<T, ArgTypes&&...>::result_type
    operator()(ArgTypes&&... args) const;
```

1 *Returns:* *INVOKE*`(get(), std::forward<ArgTypes>(args)...)`. (20.6.2)

20.6.5.5 reference_wrapper helper functions [refwrap.helpers]

```
template <ObjectType T> reference_wrapper<T> ref(T& t);
```

1 *Returns:* `reference_wrapper<T>(t)`

2 *Throws:* nothing.

```
template <ObjectType T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

3 *Returns:* `ref(t.get())`

4 *Throws:* nothing.

```
template <ObjectType T> reference_wrapper<const T> cref(const T& t);
```

5 *Returns:* `reference_wrapper <const T>(t)`

6 *Throws:* nothing.

```
template <ObjectType T> reference_wrapper<const T> cref(reference_wrapper<T> t);
```

7 *Returns:* `cref(t.get());`

8 *Throws:* nothing.

20.6.6 Identity operation [identity.operation]

```
template <IdentityOf T> struct identity {
  typedef T type;

  requires ReferentType<T>
    const T& operator()(const T& x) const;
};

requires ReferentType<T>
  const T& operator()(const T& x) const;
```

1 *Returns:* `x`

20.6.7 Arithmetic operations [arithmetic.operations]

1 The library provides basic function object classes for all of the arithmetic operators in the language (5.6, 5.7).

```
template <ReferentType T> struct plus : binary_function<T,T,T> {
  requires HasPlus<T, T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns `x + y`.

```
template <ReferentType T> struct minus : binary_function<T,T,T> {
  requires HasMinus<T,T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns `x - y`.

```
template <ReferentType T> struct multiplies : binary_function<T,T,T> {
  requires HasMultiply<T,T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

4 `operator()` returns `x * y`.

```
template <ReferentType T> struct divides : binary_function<T,T,T> {
  requires HasDivide<T,T> && Convertible<T::result_type, T>
  T operator()(const T& x, const T& y) const;
};
```

5 `operator()` returns `x / y`.

```
template <ReferentType T> struct modulus : binary_function<T,T,T> {
  requires HasModulus<T,T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

6 `operator()` returns `x % y`.

```
template <ReferentType T> struct negate : unary_function<T,T> {
  requires HasNegate<T> && Convertible<T::result_type, T>
    T operator()(const T& x) const;
};
```

7 `operator()` returns `-x`.

20.6.8 Comparisons [comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language (5.9, 5.10).

```
template <ReferentType T> struct equal_to : binary_function<T,T,bool> {
  requires HasEqualTo<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns `x == y`.

```
template <ReferentType T> struct not_equal_to : binary_function<T,T,bool> {
  requires HasNotEqualTo<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns `x != y`.

```
template <ReferentType T> struct greater : binary_function<T,T,bool> {
  requires HasGreater<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

4 `operator()` returns `x > y`.

```
template <ReferentType T> struct less : binary_function<T,T,bool> {
  requires HasLess<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

5 `operator()` returns `x < y`.

```
template <ReferentType T> struct greater_equal : binary_function<T,T,bool> {
  requires HasGreaterEqual<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

6 `operator()` returns `x >= y`.

```
template <ReferentType T> struct less_equal : binary_function<T,T,bool> {
  requires HasLessEqual<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

7 `operator()` returns `x <= y`.

8 For templates `greater`, `less`, `greater_equal`, and `less_equal`, the specializations for any pointer type yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not.

20.6.9 Logical operations [logical.operations]

1 The library provides basic function object classes for all of the logical operators in the language (5.14, 5.15, 5.3.1).

```
template <ReferentType T> struct logical_and : binary_function<T,T,bool> {
  requires HasLogicalAnd<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns `x && y`.

```
template <ReferentType T> struct logical_or : binary_function<T,T,bool> {
  requires HasLogicalOr<T, T>
    bool operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns `x || y`.

```
template <ReferentType T> struct logical_not : unary_function<T,bool> {
  requires HasLogicalNot<T>
    bool operator()(const T& x) const;
};
```

4 `operator()` returns `!x`.

20.6.10 Bitwise operations [bitwise.operations]

1 The library provides basic function object classes for all of the bitwise operators in the language (5.11, 5.13, 5.12).

```
template <ReferentType T> struct bit_and : binary_function<T,T,T> {
  requires HasBitAnd<T, T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns `x & y`.

```
template <ReferentType T> struct bit_or : binary_function<T,T,T> {
  requires HasBitOr<T, T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns `x | y`.

```
template <ReferentType T> struct bit_xor : binary_function<T,T,T> {
  requires HasBitXor<T, T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

4 `operator()` returns `x ^ y`.

20.6.11 Negators [negators]

1 Negators `not1` and `not2` take a unary and a binary predicate, respectively, and return their complements (5.3.1).

```
template <class Predicate>
  class unary_negate
    : public unary_function<typename Predicate::argument_type,bool> {
public:
  explicit unary_negate(const Predicate& pred);
  bool operator()(const typename Predicate::argument_type& x) const;
};
```

2 `operator()` returns `!pred(x)`.

```
template <class Predicate>
  unary_negate<Predicate> not1(const Predicate& pred);
```

3 *Returns:* `unary_negate<Predicate>(pred)`.

```
template <class Predicate>
  class binary_negate
    : public binary_function<typename Predicate::first_argument_type,
        typename Predicate::second_argument_type, bool> {
  public:
    explicit binary_negate(const Predicate& pred);
    bool operator()(const typename Predicate::first_argument_type&  x,
        const typename Predicate::second_argument_type& y) const;
  };
```

4 `operator()` returns `!pred(x,y)`.

```
template <class Predicate>
  binary_negate<Predicate> not2(const Predicate& pred);
```

5 *Returns:*

`binary_negate<Predicate>(pred)`.

20.6.12 Template function `bind` [bind]

1 The template function `bind` returns an object that binds a function object passed as an argument to

additional arguments.

20.6.12.1 Function object binders [func.bind]

1 This subclause describes a uniform mechanism for binding arguments of function objects.

20.6.12.1.1 Class template `is_bind_expression` [func.bind.isbind]

```
namespace std {
  template<class T> struct is_bind_expression {
    static const bool value = see below;
  };
}
```

1 `is_bind_expression` can be used to detect function objects generated by `bind`. `bind` uses `is_bind_expression` to detect subexpressions. Users may specialize this template to indicate that a type should be treated as a subexpression in a `bind` call.

```
static const bool value;
```

2 `true` if `T` is a type returned from `bind`, `false` otherwise.

20.6.12.1.2 Class template `is_placeholder` [func.bind.isplace]

```
namespace std {
  template<class T> struct is_placeholder {
    static const int value = see below;
  };
}
```

1 `is_placeholder` can be used to detect the standard placeholders `_1`, `_2`, and so on. `bind` uses `is_placeholder` to detect placeholders. Users may specialize this template to indicate a placeholder type.

```
static const int value;
```

2 `value` is J if `T` is the type of `std::placeholders::_`J, 0 otherwise.

20.6.12.1.3 Function template `bind` [func.bind.bind]

```
template<CopyConstructible F, CopyConstructible... BoundArgs>
  unspecified  bind(F f, BoundArgs... bound_args);
```

1 *Requires:* `INVOKE (f, w1, w2, ..., wN)` (20.6.2) shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(bound_args)`.

2 *Returns:* A forwarding call wrapper `g` with a weak result type (20.6.2). The effect of `g(u1, u2, ..., uM)` shall be `INVOKE(f, v1, v2, ..., vN, Callable<F cv,V1, V2, ..., VN>::result_type)`, where *cv* represents the *cv*-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below.

```
template<Returnable R, CopyConstructible F, CopyConstructible... BoundArgs>
  unspecified  bind(F f, BoundArgs... bound_args);
```

3 *Requires:* `INVOKE(f, w1, w2, ..., wN)` shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(bound_args)`.

4 *Returns:* A forwarding call wrapper `g` with a nested type `result_type` defined as a synonym for `R`. The effect of `g(u1, u2, ..., uM)` shall be *INVOKE*`(f, v1, v2, ..., vN, R)`, where the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below.

20.6.12.1.4 Placeholders [func.bind.place]

```
namespace std {
  namespace placeholders {
    // M is the implementation-defined number of placeholders
    extern unspecified  _1;
    extern unspecified  _2;
                .
                .
                .
    extern unspecified _M;
  }
}
```

1 All placeholder types shall be `DefaultConstructible` and `CopyConstructible`, and their default constructors and copy constructors shall not throw exceptions. It is implementation defined whether placeholder types are `Assignable`. `Assignable` placeholders' copy assignment operators shall not throw exceptions.

20.6.13 Adaptors for pointers to functions [function.pointer.adaptors]

1 To allow pointers to (unary and binary) functions to work with function adaptors the library provides:

```
template <CopyConstructible Arg, Returnable Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
public:
  explicit pointer_to_unary_function(Result (*f)(Arg));
  Result operator()(Arg x) const;
};
```

2 `operator()` returns `f(x)`.

```
template <CopyConstructible Arg, Returnable Result>
  pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));
```

3 *Returns:* `pointer_to_unary_function<Arg, Result>(f)`.

```
template <CopyConstructible Arg1, CopyConstructible Arg2, Returnable Result>
class pointer_to_binary_function :
  public binary_function<Arg1,Arg2,Result> {
public:
  explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
  Result operator()(Arg1 x, Arg2 y) const;
};
```

4 `operator()` returns `f(x,y)`.

```
template <CopyConstructible Arg1, CopyConstructible Arg2, Returnable Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*f)(Arg1, Arg2));
```

5 *Returns:* `pointer_to_binary_function<Arg1,Arg2,Result>(f)`.

6 [*Example:*

```
int compare(const char*, const char*);
replace_if(v.begin(), v.end(),
    not1(bind2nd(ptr_fun(compare), "abc")), "def");
```

replaces each `abc` with `def` in sequence `v`. — *end example*]

20.6.14 Adaptors for pointers to members [member.pointer.adaptors]

1 The purpose of the following is to provide the same facilities for pointer to members as those provided for pointers to functions in 20.6.13.

```
template <Returnable S, ClassType T> class mem_fun_t
        : public unary_function<T*, S> {
public:
  explicit mem_fun_t(S (T::*p)());
  S operator()(T* p) const;
};
```

2 `mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <Returnable S, ClassType T, CopyConstructible A> class mem_fun1_t
      : public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
```

3 `mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<Returnable S, ClassType T> mem_fun_t<S,T>
   mem_fun(S (T::*f)());
template<Returnable S, ClassType T, CopyConstructible A> mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A));
```

4 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <Returnable S, ClassType T> class mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*p)());
  S operator()(T& p) const;
};
```

5 `mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <Returnable S, ClassType T, CopyConstructible A> class mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*p)(A));
  S operator()(T& p, A x) const;
};
```

6 `mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<Returnable S, ClassType T> mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)());
template<Returnable S, ClassType T, CopyConstructible A> mem_fun1_ref_t<S,T,A>
   mem_fun_ref(S (T::*f)(A));
```

7 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

```
template <Returnable S, ClassType T> class const_mem_fun_t
      : public unary_function<const T*, S> {
public:
  explicit const_mem_fun_t(S (T::*p)() const);
  S operator()(const T* p) const;
};
```

8 `const_mem_fun_t` calls the member function it is initialized with given a pointer argument.

```
template <Returnable S, ClassType T, CopyConstructible A> class const_mem_fun1_t
      : public binary_function<const T*, A, S> {
public:
  explicit const_mem_fun1_t(S (T::*p)(A) const);
  S operator()(const T* p, A x) const;
};
```

9 `const_mem_fun1_t` calls the member function it is initialized with given a pointer argument and an additional argument of the appropriate type.

```
template<cReturnable S, ClassType T> const_mem_fun_t<S,T>
   mem_fun(S (T::*f)() const);
template<Returnable S, ClassType T, CopyConstructible A> const_mem_fun1_t<S,T,A>
   mem_fun(S (T::*f)(A) const);
```

10 `mem_fun(&X::f)` returns an object through which `X::f` can be called given a pointer to an `X` followed by the argument required for `f` (if any).

```
template <Returnable S, ClassType T> class const_mem_fun_ref_t
      : public unary_function<T, S> {
public:
  explicit const_mem_fun_ref_t(S (T::*p)() const);
  S operator()(const T& p) const;
};
```

11 `const_mem_fun_ref_t` calls the member function it is initialized with given a reference argument.

```
template <Returnable S, ClassType T, CopyConstructible A> class const_mem_fun1_ref_t
      : public binary_function<T, A, S> {
public:
  explicit const_mem_fun1_ref_t(S (T::*p)(A) const);
  S operator()(const T& p, A x) const;
};
```

12 `const_mem_fun1_ref_t` calls the member function it is initialized with given a reference argument and an additional argument of the appropriate type.

```
template<Returnable S, ClassType T> const_mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)() const);
template<Returnable S, ClassType T, CopyConstructible A> const_mem_fun1_ref_t<S,T,A>
    mem_fun_ref(S (T::*f)(A) const);
```

13 `mem_fun_ref(&X::f)` returns an object through which `X::f` can be called given a reference to an `X` followed by the argument required for `f` (if any).

20.6.15 Function template `mem_fn` [func.memfn]

```
template<Returnable R, class T> unspecified mem_fn(R T::* pm);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...));
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...) const);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...) volatile);
template<Returnable R, class T, CopyConstructible... Args>
  unspecified  mem_fn(R (T::* pm)(Args...) const volatile);
```

1 *Returns:* A simple call wrapper ([20.6.1]) `fn` such that the expression `fn(t, a2, ..., aN)` is equivalent to *`INVOKE`*`(pm, t, a2, ..., aN)` ([20.6.2]). `fn` shall have a nested type `result_type` that is a synonym for the return type of `pm` when `pm` is a pointer to member function.

2 The simple call wrapper shall be derived from `std::unary_function<`*`cv`* `T*, `*`Ret`*`>` when `pm` is a pointer to member function with cv-qualifier *cv* and taking no arguments, where *Ret* is `pm`'s return type.

3 The simple call wrapper shall be derived from `std::binary_function<`*`cv`* `T*, T1, `*`Ret`*`>` when `pm` is a pointer to member function with cv-qualifier *cv* and taking one argument of type `T1`, where *Ret* is `pm`'s return type.

4 *Throws:* nothing.

20.6.16 Polymorphic function wrappers [func.wrap]

1 This subclause describes a polymorphic wrapper class that encapsulates arbitrary function objects.

20.6.16.1 Class `bad_function_call` [func.wrap.badcall]

1 An exception of type `bad_function_call` is thrown by `function::operator()` (20.6.16.2.4) when the function wrapper object has no target.

```
namespace std {
  class bad_function_call : public std::exception {
  public:
    // 20.6.16.1.1, constructor:
    bad_function_call();
  };
} // namespace std
```

20.6.16.1.1 `bad_function_call` constructor [func.wrap.badcall.const]

```
bad_function_call();
```

1 *Effects:* constructs a `bad_function_call` object.

20.6.16.2 Class template `function` [func.wrap.func]

```
namespace std {
  template<FunctionType> class function; // undefined

  template<Returnable R, CopyConstructible... ArgTypes>
  class function<R(ArgTypes...)>
    : public unary_function<T1, R>      // iff sizeof...(ArgTypes) == 1 and
                                        // ArgTypes contains T1
    : public binary_function<T1, T2, R> // iff sizeof...(ArgTypes) == 2 and
                                        // ArgTypes contains T1 and T2
  {
  public:
    typedef R result_type;

    // 20.6.16.2.1, construct/copy/destroy:
    explicit function();
    function(nullptr_t);
    function(const function&);
    function(function&&);
    template<class F>
      requires CopyConstructible<F> && Callable<F, ArgTypes...>
            && Convertible<Callable<F, ArgTypes...>::result_type, R>
      function(F);
    template<class F>
      requires CopyConstructible<F> && Callable<F, ArgTypes...>
            && Convertible<Callable<F, ArgTypes...>::result_type, R>
      function(F&&);
    template<Allocator A>
      function(allocator_arg_t, const A&);
    template<Allocator A> function(allocator_arg_t, const A&,
      nullptr_t);
    template<Allocator A> function(allocator_arg_t, const A&,
      const function&);
    template<Allocator A> function(allocator_arg_t, const A&,
      function&&);
    template<class F, Allocator A> function(allocator_arg_t, const A&, F);
    template<class F, Allocator A> function(allocator_arg_t, const A&, F&&);

    function& operator=(const function&);
    function& operator=(function&&);
    function& operator=(nullptr_t);
    template<class F>
      requires CopyConstructible<F> && Callable<F, ArgTypes..>
            && Convertible<Callable<F, ArgTypes...>::result_type
      function& operator=(F);
    template<class F>
      requires CopyConstructible<F> && Callable<F, ArgTypes...>
            && Convertible<Callable<F, ArgTypes...>::result_type, R>
      function& operator=(F&&);
    template<class F>
      requires Callable<F, ArgTypes...>
            && Convertible<Callable<F, ArgTypes...>::result_type, R>
      function& operator=(reference_wrapper<F>);

    ~function();
```

```
    // 20.6.16.2.2, function modifiers:
    void swap(function&);
    template<class F, Allocator A>
      requires Callable<F, ArgTypes...>
            && Convertible<Callable<F, ArgTypes...>::result_type, R>
      void assign(F, const A&);

    // 20.6.16.2.3, function capacity:
    explicit operator bool() const;

    // deleted overloads close possible hole in the type system
    template<class R2, class... ArgTypes2>
        bool operator==(const function<R2(ArgTypes2...)>&) = delete;
    template<class R2, class... ArgTypes2>
        bool operator!=(const function<R2(ArgTypes2...)>&) = delete;

    // 20.6.16.2.4, function invocation:
    R operator()(ArgTypes...) const;

    // 20.6.16.2.5, function target access:
    const std::type_info& target_type() const;
    template <typename T>
      requires Callable<T, ArgTypes...>
            && Convertible<Callable<T, ArgTypes...>::result_type, R>
      T* target();
    template <typename T>
      requires Callable<T, ArgTypes...>
            && Convertible<Callable<T, ArgTypes...>::result_type, R>
      const T* target() const;
  };

template <class R, class... Args>
  concept_map UsesAllocator<function<R(Args...)>, Alloc> {
    typedef Alloc allocator_type;
  }

  // 20.6.16.2.6, Null pointer comparisons:
  template <MoveConstructible R, MoveConstructible... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t);

  template <MoveConstructible R, MoveConstructible... ArgTypes>
    bool operator==(nullptr_t, const function<R(ArgTypes...)>&);

  template <MoveConstructible R, MoveConstructible... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, nullptr_t);

  template <MoveConstructible R, MoveConstructible... ArgTypes>
    bool operator!=(nullptr_t, const function<R(ArgTypes...)>&);

  // 20.6.16.2.7, specialized algorithms:
  template <MoveConstructible R, MoveConstructible... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);
} // namespace std
```

20.6.16.2.1 `function` construct/copy/destroy [func.wrap.func.con]

```
explicit function();
```

1 *Postconditions:* `!*this`.

2 *Throws:* nothing.

```
function(nullptr_t);
```

3 *Postconditions:* `!*this`.

4 *Throws:* nothing.

```
function(const function& f);
```

5 *Postconditions:* `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

6 *Throws:* shall not throw exceptions if `f`'s target is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored function object. [*Note:* Implementations are encouraged to avoid the use of dynamically allocated memory for small function objects, e.g., where `f`'s target is an object holding only a pointer or reference to an object and a member function pointer. — *end note*]

```
function(function&& f);
```

7 *Effects:* If `!f`, `*this` has no target; otherwise, move-constructs the target of `f` into the target of `*this`, leaving `f` in a valid state with an unspecified value.

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function(F f);
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function(F&& f);
```

8 *Postconditions:* `!*this` if any of the following hold:

— `f` is a NULL function pointer.

— `f` is a NULL member function pointer.

— `F` is an instance of the `function` class template, and `!f`

9 Otherwise, `*this` targets a copy of `f` or `std::move(f)` if `f` is not a pointer to member function, and targets a copy of `mem_fn(f)` if `f` is a pointer to member function.

10 *Throws:* shall not throw exceptions when `f` is a function pointer or a `reference_wrapper<T>` for some `T`. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy or move constructor.

```
function& operator=(const function& f);
```

11 *Effects:* `function(f).swap(*this);`

12 *Returns:* `*this`

```
function& operator=(function&& f);
```

13 *Effects:* Replaces the target of `*this` with the target of `f`, leaving `f` in a valid but unspecified state.

14 *Returns:* `*this`

```
function& operator=(nullptr_t);
```

15 *Effects:* If `*this != NULL`, destroys the target of `this`.

16 *Postconditions:* `!(*this)`.

17 *Returns:* `*this`

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..>
        && Convertible<Callable<F, ArgTypes...>::result_type
  operator=(F f);
```

18 *Effects:* `function(f).swap(*this);`

19 *Returns:* `*this`

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..>
        && Convertible<Callable<F, ArgTypes...>::result_type
  function& operator=(F&& f);
```

20 *Effects:* Replaces the target of `*this` with `f`, leaving `f` in a valid but unspecified state. [*Note:* A valid implementation is `function(f).swap(*this)`.

21 *Returns:* `*this`.

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes...>
        && Convertible<Callable<F, ArgTypes...>::result_type, R>
  function& operator=(reference_wrapper<F> f);
```

22 *Effects:* `function(f).swap(*this);`

23 *Returns:* `*this`

24 *Throws:* nothing.

20.6.16.2.2 `function` modifiers [func.wrap.func.mod]

```
void swap(function&& other);
```

1 *Effects:* interchanges the targets of `*this` and `other`.

2 *Throws:* nothing.

```
template<class F, class A> void assign(F f, const A& a);
```

Effects: `function(f, a).swap(*this)`

20.6.16.2.3 `function` capacity [func.wrap.func.cap]

```
explicit operator bool() const
```

1 *Returns:* `true` if `*this` has a target, otherwise `false`.

2 *Throws:* nothing.

20.6.16.2.4 `function` invocation [func.wrap.func.inv]

```
R operator()(ArgTypes... args) const
```

1 *Effects:* *INVOKE*`(f, t1, t2, ..., tN, R)` (20.6.2), where `f` is the target object (20.6.1) of `*this` and `t1`, `t2`, ..., `tN` are the values in `args...`.

2 *Returns:* Nothing if `R` is `void`, otherwise the return value of *INVOKE*`(f, t1, t2, ..., tN, R)`.

3 *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped function object.

20.6.16.2.5 function target access [func.wrap.func.targ]

```
const std::type_info& target_type() const;
```

1 *Returns:* If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

2 *Throws:* nothing.

```
template<typename T>
  requires Callable<T, ArgTypes...> && Convertible<Callable<T, ArgTypes...>::result_type, R>
  T* target();
template<typename T>
  requires Callable<T, ArgTypes...> && Convertible<Callable<T, ArgTypes...>::result_type, R>
  const T* target() const;
```

3 *Returns:* If `type() == typeid(T)`, a pointer to the stored function target; otherwise a null pointer.

4 *Throws:* nothing.

20.6.16.2.6 null pointer comparison operators [func.wrap.func.nullptr]

```
template <MoveConstructible R, MoveConstructible... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>& f, nullptr_t);

template <MoveConstructible R, MoveConstructible... ArgTypes>
  bool operator==(nullptr_t, const function<R(ArgTypes...)>& f);
```

1 *Returns:* `!f`.

2 *Throws:* nothing.

```
template <MoveConstructible R, MoveConstructible... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>& f, nullptr_t);

template <MoveConstructible R, MoveConstructible... ArgTypes>
  bool operator!=(nullptr_t, const function<R(ArgTypes...)>& f);
```

3 *Returns:* `(bool) f`.

4 *Throws:* nothing.

20.6.16.2.7 specialized algorithms [func.wrap.func.alg]

```
template<Returnable R, CopyConstructible... ArgTypes>
  void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
```

1 *Effects:* `f1.swap(f2);`

20.6.17 Class template `hash` [unord.hash]

1 The unordered associative containers defined in Clause 23.4 use specializations of `hash` as the default hash function. This class template is only required to be instantiable for integer types (3.9.1), floating-point types (3.9.1), pointer types (8.3.1), and `std::string`, `std::u16string`, `std::u32string`, `std::wstring`, `std::error_code`, `std::thread::id`, `std::bitset`, and `std::vector<bool>`.

```
namespace std {
  template <class T>
  struct hash : public std::unary_function<T, std::size_t> {
    std::size_t operator()(T val) const;
  };
}
```

2 The return value of `operator()` is unspecified, except that equal arguments shall yield the same result. `operator()` shall not throw exceptions.

20.6.18 Class template `reference_closure` [func.referenceclosure]

```
namespace std {
  template<class> class reference_closure; // undefined

  template<class R , class... ArgTypes >
    class reference_closure<R (ArgTypes...)> {
    public:
      typedef R result_type;
      typedef T1 argument_type;          // iff sizeof...(ArgTypes) == 1 and ArgTypes contains T1
      typedef T1 first_argument_type;    // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1,
T2
      typedef T2 second_argument_type;   // iff sizeof...(ArgTypes) == 2 and ArgTypes contains T1,
T2

      // 20.6.18.1, construct/copy/destroy:
      reference_closure() = default;
      reference_closure(const reference_closure&) = default;
      constexpr reference_closure(nullptr_t);
      reference_closure& operator=(const reference_closure&) = default;
      reference_closure& operator=(nullptr_t);
      ~reference_closure() = default;

      // 20.6.18.2, observer:
      explicit operator bool() const;

      // 20.6.18.3, invocation:
      R operator()(ArgTypes...) const;
  };

  // 20.6.18.4, comparisons:
  template <class R, class... ArgTypes>
    bool operator==(const reference_closure<R (ArgTypes...)>&, nullptr_t);
  template <class R, class... ArgTypes>
    bool operator==(nullptr_t, const reference_closure<R (ArgTypes...)>&);
  template <class R, class... ArgTypes>
```

```
    bool operator!=(const reference_closure<R(ArgTypes...)>&, nullptr_t);
  template <class R, class... ArgTypes>
    bool operator!=(nullptr_t, const reference_closure<R (ArgTypes...)>&);
}
```

1 The `reference_closure` class template represents reference-only closures (5.1.1).

2 A `reference_closure` object `f` of type `F` is Callable for argument types `T1, T2, ..., TN` in `ArgTypes` and a return type `R` if, given lvalues `t1, t2, ..., tN` of types `T1, T2, ..., TN`, respectively, *INVOKE* `(f, t1, t2, ..., tN)` is well formed and, if `R` is not `void`, convertible to `R`.

3 The instances of `reference_closure` class template are trivial and standard-layout classes (3.9).

4 Unless otherwise specified, the functions in this section do not throw exceptions.

20.6.18.1 Construct, copy, destroy [func.referenceclosure.cons]

`reference_closure();`

1 *Postcondition:* None — the object state is undefined.

`reference_closure(const reference_closure& f)`

2 *Postcondition:* `*this` is a copy of `f`.

`reference_closure(nullptr_t)`

3 *Postcondition:* `!*this`

`reference_closure& operator=(const reference_closure& f)`

4 *Postcondition:* `*this` is a copy of `f`.

5 *Returns:* `*this`.

`reference_closure& operator=(nullptr_t);`

6 *Postcondition:* `!*this`

7 *Returns:* `*this`

`~reference_closure();`

8 *Effects:* destroys the object.

20.6.18.2 Observer [func.referenceclosure.obs]

`explicit operator bool() const;`

1 *Returns:* `true` if `*this` was constructed or copied from a closure, `false` if `*this` was constructed or copied from an object of type `nullptr_t`, undefined otherwise.

20.6.18.3 Invocation [func.referenceclosure.invoke]

`R operator()(ArgTypes... args) const;`

1 *Effects:* Undefined if `*this` was default constructed, constructed from an object of type `nullptr_t` or copied from such. Otherwise, invokes the closure with the given arguments.

2 *Returns:* Nothing if `R` is `void`, otherwise the return value of the closure.

3 *Throws:* Any exception thrown by the wrapped function object.

20.6.18.4 Comparison [func.referenceclosure.compare]

```
template <class R, class... ArgTypes>
  bool operator==(const reference_closure<R(ArgTypes...)>& f, nullptr_t);
template <class R, class... ArgTypes>
  bool operator==(nullptr_t, const reference_closure<R(ArgTypes...)>& f;
```

1 *Returns:* `!f`

```
template <class R, class... ArgTypes>
  bool operator!=(const reference_closure<R(ArgTypes...)>& f, nullptr_t);
template <class R, class... ArgTypes>
  bool operator!=(nullptr_t, const reference_closure<R(ArgTypes...)>& f;
```

2 *Returns:* `(bool)f`

20.7 Memory [memory]

1 **Header <memory_concepts> synopsis**

```
namespace std {
  // 20.7.2.2 Allocator concepts
  auto concept Allocator<Typename Alloc> see below;

  // 20.7.3 Allocator element concepts
  auto concept HasAllocatorType<class T> see below;
  auto concept UsesAllocator<class T, class Alloc> see below;

  concept ConstructibleWithAllocator<class T, class Alloc, class... Args> see below;
  template <Allocator alloc, class T, class... Args>
    requires see below
    concept_map ConstructibleWithAllocator<T, Alloc, Args&&...> see below;

  concept AllocatableElement<class Alloc, class T, class... Args> see below;

  template <Allocator Alloc, class T, class... Args>
    requires HasConstructor<T, Args&&...>
    concept_map AllocatableElement<Alloc, T, Args&&...> see below;
}
```

2 **Header <memory> synopsis**

```
namespace std {
  // 20.7.1, allocator argument tag
  struct allocator_arg_t { };
  const allocator_arg_t allocator_arg = allocator_arg_t();

  // 20.7.3, allocator-related traits
  template <class Alloc> struct is_scoped_allocator;

  // 20.7.4, allocation propagation traits
  template <class Alloc> struct allocator_propagate_never;
  template <class Alloc> struct allocator_propagate_on_copy_construction;
  template <class Alloc> struct allocator_propagate_on_move_assignment;
```

```
template <class Alloc> struct allocator_propagate_on_copy_assignment;
template <class Alloc> struct allocator_propagation_map;

// 20.7.6, the default allocator:
template <class T> class allocator;
template <ObjectType T>
  concept_map Allocator<allocator<T> > { };
template <> class allocator<void>;
template <class T, class U>
  bool operator==(const allocator<T>&, const allocator<U>&) throw();
template <class T, class U>
  bool operator!=(const allocator<T>&, const allocator<U>&) throw();

// 20.7.7, scoped allocator adaptor
template <Allocator OuterA, Allocator InnerA = unspecified allocator type>
  class scoped_allocator_adaptor;
template <Allocator Alloc>
  class scoped_allocator_adaptor<Alloc, unspecified allocator type>;
template <Allocator OuterA, Allocator InnerA>
  struct is_scoped_allocator<scoped_allocator_adaptor<OuterA, InnerA> >
    : true_type { };
template <Allocator OuterA, Allocator InnerA>
  struct allocator_propagate_never<scoped_allocator_adaptor<OuterA, InnerA> >
    : true_type { };
template <Allocator OuterA1, Allocator OuterA2, Allocator InnerA>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerA1>& a,)
                  const scoped_allocator_adaptor<OuterA2, InnerA>& b);
template <Allocator OuterA1, Allocator OuterA2, Allocator InnerA>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerA1>& a,)
                  const scoped_allocator_adaptor<OuterA2, InnerA>& b);

// 20.7.8, raw storage iterator:
template <class OutputIterator, class T> class raw_storage_iterator;

// 20.7.9, temporary buffers:
template <class T>
  pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
  void return_temporary_buffer(T* p);

// 20.7.10, construct element
template <Allocator Alloc, class T, class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void construct_element(Alloc& alloc, T& r, Args&&... args);

// 20.7.11, specialized algorithms:
template <ObjectType T> T* addressof(T& r);
template <ObjectType T> T* addressof(T&& r);
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
template <class ForwardIterator, class T>
```

```
  void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                          const T& x);
template <class ForwardIterator, class Size, class T>
  void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);

// 20.7.12 Class unique_ptr:
template <class X> class default_delete;
template <class X, class D = default_delete<T>> class unique_ptr;

// 20.7.13.1, Class bad_weak_ptr:
class bad_weak_ptr;

// 20.7.13.2, Class template shared_ptr:
template<class T> class shared_ptr;

// 20.7.13.2.7, shared_ptr comparisons:
template<class T, class U>
  bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U>
  bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U>
  bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);
template <class T, class U>
  bool operator>(shared_ptr<T> const& a, shared_ptr<U> const& b);
template <class T, class U>
  bool operator<=(shared_ptr<T> const& a, shared_ptr<U> const& b);
template <class T, class U>
  bool operator>=(shared_ptr<T> const& a, shared_ptr<U> const& b);

// 20.7.13.2.9, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// 20.7.13.2.10, shared_ptr casts:
template<class T, class U>
  shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
template<class T, class U>
  shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

// 20.7.13.2.8, shared_ptr I/O:
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

// 20.7.13.2.11, shared_ptr get_deleter:
template<class D, class T> D* get_deleter(shared_ptr<T> const& p);

// 20.7.13.3, Class template weak_ptr:
template<class T> class weak_ptr;

// 20.7.13.3.6, weak_ptr specialized algorithms:
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);

// 20.7.13.4 Class template owner_less
template <class T> class owner_less;
```

```
  // 20.7.13.5, Class enable_shared_from_this:
  template<class T> class enable_shared_from_this;

  // 20.7.13.6, shared_ptr atomic access:
  template<class T>
    bool atomic_is_lock_free(const shared_ptr<T>* p);

  template<class T>
    shared_ptr<T> atomic_load(const shared_ptr<T>* p);
  template<class T>
    shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);

  template<class T>
    void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
    void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);

  template<class T>
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                           memory_order mo);

  template<class T>
    bool atomic_compare_exchange_weak(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_strong(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_weak_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);
  template<class T>
    bool atomic_compare_exchange_strong_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);

  // 20.7.13.7, Pointer safety
  enum class pointer_safety { relaxed, preferred, strict };
  void declare_reachable(void *p);
  template <class T> T *undeclare_reachable(T *p);
  void declare_no_pointers(char *p, size_t n);
  void undeclare_no_pointers(char *p, size_t n);
  pointer_safety get_pointer_safety();

  // 20.7.14, Pointer alignment function
  void *align(std::size_t alignment, std::size_t size,
    void *&ptr, std::size_t& space);
}
```

20.7.1 Allocator argument tag [allocator.tag]

```
namespace std {
```

```
  struct allocator_arg_t { };
  const allocator_arg_t allocator_arg = allocator_arg_t();
}
```

1 The `allocator_arg_t` struct is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, several types (see `pair`, 20.2.3) have constructors with `allocator_arg_t` as the first argument, immediately followed by an argument of a type that satisfies the `Allocator` requirements (20.7.2.2).

20.7.2 Allocators [allocator]

20.7.2.1 In general [allocator.general]

1 The library describes a standard set of requirements for allocators, which are objects that encapsulate the information about an allocation model. This information includes the knowledge of pointer types, the type of their difference, the type of the size of objects in this allocation model, as well as the memory allocation and deallocation primitives for the model. All of the string types (21) and containers (23) are parametrized in terms of allocators.

2 If the alignment associated with a specific over-aligned type is not supported by an allocator, instantiation of the allocator for that type may fail. The allocator also may silently ignore the requested alignment. [*Note:* additionally, the member function `allocate` for that type may fail by throwing an object of type `std::bad_alloc`. — *end note*]

20.7.2.2 Allocator concept [allocator.concepts]

```
auto concept Allocator<typename X> :
  CopyConstructible<X>, EqualityComparable<X> {
    ObjectType value_type = typename X::value_type;
    Dereferenceable pointer = see below;
    Dereferenceable const_pointer = see below;
    requires Regular<pointer>
        && RandomAccessIterator<pointer>
        && Regular<const_pointer>
        && RandomAccessIterator<const_pointer>;
    SignedIntegralLike difference_type =
      RandomAccessIter<pointer>::difference_type;
    typename generic_pointer = void*;
    typename const_generic_pointer = const void*;
    typename reference = value_type&;
    typename const_reference = const value_type&;
    UnsignedIntegralLike size_type = see below;
    template <ObjectType T> class rebind = see below;

    requires Destructible<value_type>;
    requires Convertible<pointer, const_pointer>
        && Convertible<pointer, generic_pointer>
        && SameType<pointer::reference, value_type&>
        && SameType<pointer::reference, reference>;
    requires Convertible<const_pointer, const_generic_pointer>
        && SameType<const_pointer::reference, const value_type&>
        && Sametype<const_pointer::reference, const_reference>;
    requires Sametype<rebind<value_type>, X>;
    requires SameType<generic_pointer,
      rebind<unspecified unique type>::generic_pointer>;
    requires SameType<const_generic_pointer,
```

```
    rebind<unspecified unique type>::const_generic_pointer>;

  pointer X::allocate(size_type n);
  pointer X::allocate(size_type n, const_generic_pointer p);
  void X::deallocate(pointer p, size_type n);
  size_type X::max_size() const { return numeric_limits<size_type>::max(); }

  template <ObjectType T> X::X(const rebind<T>& y);

  template <class... Args>
    requires Hasconstructor<value_type, Args&&...>
    void X::construct(value_type* p, Args&&... args) {
      ::new ((void*)p) value_type(forward<Args>(args)...);
    }
  void X::destroy(value_type* p) {
    addressof(*p)->~value_type();
  }
  pointer X::addressof(reference r) const {
    return addressof(r);  // see below
  }
  const_pointer X::addressof(const_reference r) const {
    return addressof(r);  // see below
  }
}
```

`ObjectType value_type;`

1 *Type:* the type of objects allocated by `X`.

`Dereferenceable pointer;`
`Dereferenceable const_pointer;`

2 *Type:* a pointer-like (const pointer-like) type used to refer to memory allocated by objects of type `X`. The default `pointer` type is `X::pointer` if such a type is declared and `value_type*` otherwise. The default `const_pointer` type is `X::const_pointer` if such a type is declared and `const value_type*` otherwise. The behavior is undefined if an exception is propagated when applying any operation from the `Regular` concept to a `pointer`, `const_pointer`, `generic_pointer`, or `const_generic_pointer`.

`SignedIntegralLike difference_type;`

3 *Type:* a type that can represent the difference between any two pointers in the allocation model.

`typename generic_pointer;`
`typename const_generic_pointer;`

4 *Type:* a type that can store the value of a `pointer` (`const_pointer`) from any allocator in the same family (see member template `rebind` in 20.7.2.1) as `X` and which will produce the same value when explicitly converted back to that pointer type. For any two allocators `X` and `Y` of the same family, the implementation of a library facility using `Allocator<X>` and `Allocator<Y>` may add the additonal requirements `SameType<Allocator<X>::generic_pointer, Allocator<Y>::generic_pointer>` and `SameType<Allocator<X>::const_generic_pointer, Allocator<Y>::const_generic_pointer>`. [*Example:*

```
template <ObjectType T, Allocator Alloc = allocator<T> >
  requires Destructible<T> &&
  SameType<Alloc::generic_pointer,
```

```
    Alloc::Rebind<list_node<T>>::generic_pointer> &&
  SameType<Alloc::const_generic_pointer,
    Alloc::Rebind<list_node<T>>::const_generic_pointer> &&
  class list;
```

— *end example*]

```
typename reference;
typename const_reference;
```

5 *Type:* a reference (const reference) to a `value_type` object.

```
UnsignedIntegralLike size_type;
```

6 *Type:* a type that can represent the size of the largest object in the allocation model. The default `size_type` is `X::size_type` if such a type is declared and `std::size_t` otherwise.

```
template <ObjectType T> class rebind;
```

7 *Class template:* instantiations of the template are allocators in the same *family* as `X`, that is, if the name `X` is bound to `SomeAllocator<value_type>`, then `rebind<U>` is the same type as `SomeAllocator<U>`. The resulting type `SomeAllocator<U>` shall meet the requirements of the `Allocator` concept. The default template for `rebind` is a template `R` for which `R<U>` is X::template `rebind<U>::other`.

```
pointer X::allocate(size_type n);
pointer X::allocate(size_type n, const_generic_pointer hint);
```

8 *Effects:* contiguous memory is allocated for `n` objects of type `value_type` but the objects are not constructed.[226]

9 *Returns:* a pointer to the allocated memory. [*Note:* if `n == 0`, the return value is unspecified. — *end note*]

10 *Remark:* the meaning of `hint` is not specified, but it should be used as an aid to locality. [*Note:* in a container member function, a pointer to an adjacent element is often a good choice for the `hint` argument. — *end note*]

```
void X::deallocate(pointer p, size_type n);
```

11 *Preconditions:* `p` shall be a non-singular pointer value obtained from a call to `allocate()` on this allocator or one that compares equal to it. `p` shall not have been passed to `deallocate()` since the call to `allocate()`. `n` shall equal the value passed as the first argument in the call to `allocate()`. All `n value_type` objects in the area pointed to by `p` shall have been destroyed prior to this call.

12 *Effects:* deallocates the storage referenced by `p`.

13 *Throws:* nothing.

```
size_type X::max_size();
```

14 *Returns:* the largest value that can meaningfully be passed to `X::allocate()`.

```
template <class... Args>
  requires Hasconstructor<value_type, Args&&...>
```

226) `a.allocate` should be an efficient means of allocating a single object of type `T`, even when `sizeof(T)` is small. That is, there is no need for a container to maintain its own free list.

```
void X::construct(value_type* p, Args&&... args);
```

Effects: calls the constructor for the object at `p`, using the constructor arguments `args`.

Default behavior: `::new ((void*)p) value_type(forward<Args>(args)...);`

```
void X::destroy(value_type* p);
```

Effects: calls the destructor on the object at `p` but does not deallocate it.

Default behavior: `p->~value_type();`

```
pointer X::address(reference r) const;
const_pointer X::address(const_reference r) const;
```

Precondition: `r` is a reference to an object that was allocated from this allocator or one that compares equal to it.

Returns: a `pointer` to the object referred to by `r`. This concept defines a default implementation of `address` only if `pointer` is the same as `value_type*`.

20.7.3 Allocator-related element concepts [allocator.element.concepts]

```
auto concept HasAllocatorType<typename T> {
  typename allocator_type = T::allocator_type;
  requires Allocator<allocator_type>;
}
```

Remark: automatically detects whether `T` has a nested `allocator_type` that meets the requirements of an allocator.

```
auto concept UsesAllocator<typename T, typename Alloc> {
  requires Allocator<alloc>;
  typename allocator_type = T::allocator_type;
  requires Allocator<allocator_type>
      && Convertible<Alloc, allocator_type>;
}
```

Remark: Automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`. A program may define a concept map `UsesAllocator<T, Alloc>` for a user-defined type `T` that does not have a nested `allocator_type` but is nonetheless constructible using the specified `Alloc`. [*Note:* although the default concept maps for the concepts `UsesAllocator` and `HasAllocatorType` often cause them to appear in pairs, there is no inherent relationship between `UsesAllocator` and `HasAllocatorType`, nor between `!UsesAllocator` and `!HasAllocatorType`. — *end note*]

```
template <class Alloc> struct is_scoped_allocator : false_type { };
```

[*Note:* If a specialization `is_scoped_allocator<Alloc>` is derived from true_type, it indicates that `Alloc` is a scoped allocator. A scoped allocator specifies the memory resource to be used by a container (as all allocators do) and also specifies an inner allocator resource to be used by every element of the container. — *end note*]

Requires: If a specialization `is_scoped_allocator<Alloc>` is derived from `true_type`, `Alloc` shall have a nested type `inner_allocator_type` and a member function `inner_allocator()` which is callable with no arguments and which returns an object of a type that is convertible to `inner_allocator_type`.

```
concept ConstructibleWithAllocator<class T, class Alloc, class... Args> {
  T::T(allocator_arg_t, Alloc, Args&&);
}
```

[*Note:* The `ConstructibleWithAllocator` concept provides a uniform interface for passing an allocator to an object's constructor. — *end note*]

The library shall define concept map templates to adapt `ConstructibleWithAllocator` for each pattern of constraints in Table 42. Each concept map shall adapt `T`'s constructor, mapping the variadic argument pack from its position in the `ConstructibleWithAllocator` concept into its corresponding position in the actual constructor for `T` and mapping the `Alloc` and `allocator_arg_t` arguments to their appropriate positions, if any, in the argument list for `T`'s constructor. The concept maps shall be constrained such that, in situations where a set of types matches more than one pattern, the partial ordering of concept maps gives precedence to those patterns described earlier in the table. [*Note:* there are concept maps to encompass almost all types, including those that don't use allocators at all. However, there is no concept map in this library for a type that uses an allocator but does not support passing the specified allocator to the specified constructor. The last restriction prevents the allocator being quitely ignored in a context where the user is likely to expect it to be used. — *end note*]

Table 42 — `ConstructibleWithAllocator` concept map constraint patterns

Concept requirements	Constructor requirement
`UsesAllocator<T, Alloc>`	`T::T(allocator_arg_t, Alloc, Args&&...)`
`UsesAllocator<T, Alloc>`	`T::T(Args&&..., Alloc)`
`!HasAllocatorType<T> && !UsesAllocator<T, Alloc>`	`T::T(Args&&...)`

The `AllocatableElement` concept provides a uniform interface (see 20.7.10) for constructing an object from an allocator. A concept map provides a default implementation that is suitable for most allocators. Specific allocator templates may provide more specialized concept maps (for example, 20.7.7.) [*Note:* `ConstructibleWithAllocator` describes how to construct an item that uses an allocator; `AllocatableElement` describes how to consturct an item that was allocated from an allocator. — *end note*]

```
concept AllocatableElement<class Alloc, class T, class... Args> {
  requires Allocator<Alloc>;
  void construct_element(Alloc&, T*, Args&&...);
}

template <Allocator Alloc, class T, class ... Args>
  requires HasConstructor<T, Args...>
  concept_map AllocatableElement<Alloc, T, Args&&...> {
    void construct_element(Alloc& a, T* t, Args&&... args) {
      Alloc::rebind<T>(a).construct(t, forward(args)...);
    }
  }
```

20.7.4 Allocator propagation traits [allocator.propagation]

```
template <class Alloc> struct allocator_propagate_never
  : false_type { };
```

1 *Requires:* `Alloc` shall be an `Allocator` (20.7.2.2).

2 [*Note:* If specialized to derive from `true_type` for a specific allocator type, indicates that a container using the specified `Alloc` should not copy or move the allocator when the container is copy-constructed, move-constructed, copy-assigned, moved-assigned, or swapped. — *end note*]

```
template <class Alloc> struct allocator_propagate_on_copy_construction
  : false_type { };
```

3 *Requires:* `Alloc` shall be an `Allocator` (20.7.2.2).

4 [*Note:* If specialized to derive from `true_type` for specific allocator type, indicates that a container using the specified `Alloc` should copy or move the allocator when the container is copy constructed or move constructed, but not when the container is copy assigned, moved assigned, or swapped. — *end note*]

5 *Default behavior:* The unspecialized trait derives from `true_type` if none of `allocator_propagate_never`, `allocator_propagate_on_move_assignment`, or `allocator_propagate_on_copy_assignment` is derived from `true_type` for the given type `Alloc`. Otherwise, it derives from `false_type`.

```
template <class Alloc> struct allocator_propagate_on_move_assignment
  : false_type { };
```

6 *Requires:* `Alloc` shall be an `Allocator` (20.7.2.2).

7 [*Note:* if specialized to derive from `true_type` for specific allocator type, indicates that a container using the specified `Alloc` should copy or move the allocator when the container is copy constructed, move constructed, move assigned, or swapped but not when the container is copy assigned. — *end note*]

```
template <class Alloc> struct allocator_propagate_on_copy_assignment
  : false_type { };
```

8 *Requires:* `Alloc` shall be an `Allocator` (20.7.2.2).

9 [*Note:* If specialized to derive from `true_type` for a specific allocator type, indicates that a container using the specified `Alloc` should copy or move the allocator when the container is copy constructed, move constructed, move assigned, swapped or copy assigned. — *end note*]

20.7.5 Allocator propagation map [allocator.propagation.map]

```
template <class Alloc> struct allocator_propagation_map {
  static Alloc select_for_copy_construction(const Alloc&);
  static void move_assign(Alloc& to, Alloc&& from);
  static void copy_assign(Alloc& to, Alloc& from);
  static void swap(Alloc& a, Alloc& b);
}
```

1 *Requires:* Exactly one propagation trait shall derive from `true_type` for `Alloc`.

2 [*Note:* The `allocator_propagation_map` provides functions to be used by containers for manipulating allocators during construction, assignment, and swap operations. The implementations of the functions above are dependent on the allocator propagation traits of the specific `Alloc`. — *end note*]

```
static Alloc select_for_copy_construction(const Alloc&);
```

3 *Returns:* `Alloc()` if `allocator_propagate_never<Alloc>::value` is `true`, otherwise `x`.

```
static void move_assign(Alloc& to, Alloc&& from);
```

4 *Effects:* If `allocator_propagate_on_move_assignment<Alloc>::value` is `true` or if `allocator_propa- gate_on_copy_assignment<Alloc>::value` is `true`, assigns `to = forward(from)`; otherwise does nothing.

```
static void copy_assign(Alloc& to, Alloc& from);
```

Effects: If `allocator_propagate_on_copy_assignment<Alloc>::value` is `true`, assigns `to = from`; otherwise does nothing.

```
static void swap(Alloc& a, Alloc& b);
```

5 *Effects:* If `allocator_propagate_on_move_assignment<Alloc>::value` is `true` or if `allocator_propa- gate_on_copy_assignment<Alloc>::value` is `true`, exchanges the values of `a` and `b`; otherwise, if `a == b`, does nothing; otherwise the behavior is undefined.

20.7.6 The default allocator [default.allocator]

```
namespace std {
  template <class T> class allocator;

  // specialize for void:
  template <> class allocator<void> {
  public:
    typedef void*   pointer;
    typedef const void* const_pointer;
    // reference-to-void members are impossible.
    typedef void  value_type;
    template <class U> struct rebind { typedef allocator<U> other; };
  };

  template <class T> class allocator {
   public:
    typedef size_t    size_type;
    typedef ptrdiff_t difference_type;
    typedef T*        pointer;
    typedef const T*  const_pointer;
    typedef T&        reference;
    typedef const T&  const_reference;
    typedef T         value_type;
    template <class U> struct rebind { typedef allocator<U> other; };

    allocator() throw();
    allocator(const allocator&) throw();
    template <class U> allocator(const allocator<U>&) throw();
   ~allocator() throw();

    pointer address(reference x) const;
    const_pointer address(const_reference x) const;

    pointer allocate(
      size_type, allocator<void>::const_pointer hint = 0);
    void deallocate(pointer p, size_type n);
    size_type max_size() const throw();
```

```
    template<class... Args> void construct(pointer p, Args&&... args);
    void destroy(pointer p);
  };
}
```

20.7.6.1 allocator members [allocator.members]

1 Except for the destructor, member functions of the default allocator shall not introduce data races (1.10) as a result of concurrent calls to those member functions from different threads. Calls to these functions that allocate or deallocate a particular unit of storage shall occur in a single total order, and each such deallocation call shall happen before the next allocation (if any) in this order.

```
pointer address(reference x) const;
```

2 *Returns:* The actual address of the object referenced by `x`, even in the presence of an overloaded operator&.

```
const_pointer address(const_reference x) const;
```

3 *Returns:* The actual address of the object referenced by `x`, even in the presence of an overloaded operator&.

```
pointer allocate(size_type n, allocator<void>::const_pointer hint=0);
```

4 [*Note:* In a container member function, the address of an adjacent element is often a good choice to pass for the `hint` argument. — *end note*]

5 *Returns:* a pointer to the initial element of an array of storage of size `n * sizeof(T)`, aligned appropriately for objects of type `T`. It is implementation-defined whether over-aligned types are supported (3.11).

6 *Remark:* the storage is obtained by calling `::operator new(std::size_t)` (18.5.1), but it is unspecified when or how often this function is called. The use of `hint` is unspecified, but intended as an aid to locality if an implementation so desires.

7 *Throws:* `bad_alloc` if the storage cannot be obtained.

```
void deallocate(pointer p, size_type n);
```

8 *Requires:* `p` shall be a pointer value obtained from `allocate()`. `n` shall equal the value passed as the first argument to the invocation of allocate which returned `p`.

9 *Effects:* Deallocates the storage referenced by `p` .

10 *Remarks:* Uses `::operator delete(void*)` (18.5.1), but it is unspecified when this function is called.

```
size_type max_size() const throw();
```

11 *Returns:* the largest value N for which the call `allocate(N,0)` might succeed.

```
template <class... Args> void construct(pointer p, Args&&... args);
```

12 *Effects:* `::new((void *)p) T(std::forward<Args>(args)...)`

```
void destroy(pointer p);
```

13 *Effects:* `p->~T()`

20.7.6.2 `allocator` globals [allocator.globals]

```
template <class T1, class T2>
  bool operator==(const allocator<T1>&, const allocator<T2>&) throw();
```

1 *Returns:* `true`.

```
template <class T1, class T2>
  bool operator!=(const allocator<T1>&, const allocator<T2>&) throw();
```

2 *Returns:* `false`.

20.7.7 Scoped allocator adaptor [allocator.adaptor]

1 The `scoped_allocator_adaptor` class template is an allocator template that specifies the memory resource (the outer allocator) to be used by a container (as any other allocator does) and also specifies an inner allocator resource to be used by every element in the container. This adaptor is instantiated with outer and inner allocator types. If instantiated with only one allocator type (i.e., the second type is `void`), the same allocator type is used for both the outer and inner allocator types and the same allocator instance is used for both the outer and inner allocator instances. The interface is specialized for the single-allocator case such that it takes only one allocator instance argument in the constructor, versus two allocators for the general case. Otherwise, the interface to the specialized and general cases are the same. A `scoped_allocator_adaptor` that is instantiated with two identical parameters is different than an adaptor instantiated with only one parameter: the former may be constructed with different instances of outer and inner allocators whereas the second may be constructed only with one allocator instance. [*Note:* The `scoped_allocator_adaptor` is derived from the outer allocator type so it can be substituted for the outer allocator type in most expressions. —*end note*]

```
namespace std {
  template<Allocator OuterA, Allocator InnerA = unspecified allocator type>
    class scoped_allocator_adaptor;
   template<Allocator OuterA>
     class scoped_allocator_adaptor<OuterA, unspecified allocator type> : public OuterA  {
  public:
    // outer and inner allocator types are the same:
    typedef OuterA outer_allocator_type;
    typedef OuterA inner_allocator_type;

    typedef typename outer_allocator_type::size_type               size_type;
    typedef typename outer_allocator_type::difference_type         difference_type;
    typedef typename outer_allocator_type::pointer                 pointer;
    typedef typename outer_allocator_type::const_pointer           const_pointer;
    typedef typename outer_allocator_type::generic_pointer         generic_pointer;
    typedef typename outer_allocator_type::const_generic_pointer   const_generic_pointer;
    typedef typename outer_allocator_type::reference               reference;
    typedef typename outer_allocator_type::const_reference         const_reference;
    typedef typename outer_allocator_type::value_type              value_type;

    template <ObjectType U>
    struct rebind {
      typedef scoped_allocator_adaptor<
        Allocator<OuterA>::rebind<U>, unspecified allocator type> other;
    };

    scoped_allocator_adaptor();
```

```
  scoped_allocator_adaptor(scoped_allocator_adaptor&&);
  scoped_allocator_adaptor(const scoped_allocator_adaptor&);
  scoped_allocator_adaptor(OuterA&& outerAlloc);
  scoped_allocator_adaptor(const OuterA& outerAlloc);

  template <Allocator OuterA2>
    requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(scoped_allocator_adaptor<OuterA2, void>&&);
  template <Allocator OuterA2>
    requires Convertible<const OuterA2&, OuterA>
    scoped_allocator_adaptor(const scoped_allocator_adaptor<OuterA2, void>&);

 ~scoped_allocator_adaptor();

  pointer       address(reference x)       const;
  const_pointer address(const_reference x) const;

  pointer allocate(size_type n);
  pointer allocate(size_type n, const_generic_pointer u);
  void deallocate(pointer p, size_type n);
  size_type max_size() const;
  template <class... Args>
    requires HasConstructor<value_type, Args&&...>
    void construct(pointer p, Args&&... args);
  void destroy(pointer p);

  const outer_allocator_type& outer_allocator();
  const inner_allocator_type& inner_allocator();
};

template<typename OuterA, typename InnerA>
  class scoped_allocator_adaptor : public OuterA {
public:
  typedef OuterA outer_allocator_type;
  typedef InnerA inner_allocator_type;

  typedef typename outer_allocator_type::size_type               size_type;
  typedef typename outer_allocator_type::difference_type         difference_type;
  typedef typename outer_allocator_type::pointer                 pointer;
  typedef typename outer_allocator_type::const_pointer           const_pointer;
  typedef typename outer_allocator_type::generic_pointer         generic_pointer;
  typedef typename outer_allocator_type::const_generic_pointer   const_generic_pointer;
  typedef typename outer_allocator_type::reference               reference;
  typedef typename outer_allocator_type::const_reference         const_reference;
  typedef typename outer_allocator_type::value_type              value_type;

  template <ObjectType U>
  struct rebind {
    typedef scoped_allocator_adaptor<
      Allocator<OuterA>::rebind<U>, InnerA> other;
  };

  scoped_allocator_adaptor();
  scoped_allocator_adaptor(outer_allocator_type&& outerA,
                           inner_allocator_type&& innerA);
```

```
  scoped_allocator_adaptor(const outer_allocator_type& outerA,
                           const inner_allocator_type& innerA);
  scoped_allocator_adaptor(scoped_allocator_adaptor&& other);
  scoped_allocator_adaptor(const scoped_allocator_adaptor& other);

  template <Allocator OuterA2>
    requires Convertible<OuterA2&&, OuterA>
    scoped_allocator_adaptor(
      scoped_allocator_adaptor<OuterA2&,InnerA>&&);
  template <Allocator OuterA2>
    requires Convertible<const OuterA2&, OuterA>
    scoped_allocator_adaptor(
      const scoped_allocator_adaptor<OuterA2&,InnerA>&);

 ~scoped_allocator_adaptor();

  pointer       address(reference x)       const;
  const_pointer address(const_reference x) const;

  pointer allocate(size_type n);
  pointer allocate(size_type n, const_generic_pointer u);
  void deallocate(pointer p, size_type n);
  size_type max_size() const;

  template <class... Args>
    requires HasConstructor<value_type, Args&&...
    void construct(value_type* p, Args&&... args);
  void destroy(value_type* p);

  const outer_allocator_type& outer_allocator() const;
  const inner_allocator_type& inner_allocator() const;
};

template<Allocator OuterA1, Allocator OuterA2, Allocator InnerA>
bool operator==(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
                const scoped_allocator_adaptor<OuterA2,InnerA>& b);

template<Allocator OuterA1, Allocator OuterA2, Allocator InnerA>
bool operator!=(const scoped_allocator_adaptor<OuterA1,InnerA>& a,
                const scoped_allocator_adaptor<OuterA2,InnerA>& b);
}
```

20.7.7.1 `scoped_allocator_adaptor` constructors [allocator.adaptor.cntr]

```
scoped_allocator_adaptor();
```

1 *Effects:* Initializes the outer and inner allocator instances using their corresponding default constructors.

```
scoped_allocator_adaptor(scoped_allocator_adaptor&& other);
scoped_allocator_adaptor(const scoped_allocator_adaptor& other);
```

2 *Effects:* Initializes the outer and inner allocator instances from the corresponding parts of `other`.

```
scoped_allocator_adaptor(OuterA&& outer);
scoped_allocator_adaptor(const OuterA& outer);
```

3 *Requires:* `scoped_allocator_adaptor` was instantiated with only one parameter.

4 *Effects:* Initializes the base class (which is both the outer and inner allocator) from `outer`.

```
template <Allocator OuterA2>
  requires Convertible<OuterA2&&, OuterA>
  scoped_allocator_adaptor(
    scoped_allocator_adaptor<OuterA2, InnerA>&& other);
template <Allocator OuterA2>
  requires Convertible<OuterA2&&, OuterA>
  scoped_allocator_adaptor(
    const scoped_allocator_adaptor<OuterA2, InnerA>& other);
```

5 *Requires:* `Same<OuterA2, OuterA::rebind<value_type>::other>`.

6 *Effects:* Initializes the outer and inner allocator instances from the corresponding parts of `other`.

20.7.7.2 `scoped_allocator_adaptor` members [allocator.adaptor.members]

```
pointer address(reference x) const;
const_pointer address(const_reference x) const
```

1 *Returns:* `outer_allocator().address(x)`

```
pointer allocate(size_type n);
```

2 *Returns:* `outer_allocator().allocate(n)`

```
template <typename HintP>
  pointer allocate(size_type n, HintP u);
```

3 *Returns:* `outer_allocator().allocate(n, u)`

```
void deallocate(pointer p, size_type n);
```

4 *Effects:* `outer_allocator().deallocate(p, n)`

```
size_type max_size() const;
```

5 *Returns:* `outer_allocator().max_size()`

```
template <class... Args>
  requires HasConstructor<value_type, Args&&...>
  void construct(value_type* p, Args&&... args);
```

6 *Effects:* `outer_allocator().construct(p, forward<Args>(`args`)...)`

```
void destroy(pointer p)
```

7 *Effects:* `outer_allocator().destroy(p)`

```
const outer_allocator_type& outer_allocator() const;
```

8 *Returns:* The outer allocator used to construct this object.

```
const inner_allocator_type& inner_allocator() const;
```

9 *Returns:* The inner allocator used to construct this object. For the single-parameter instantiation, returns the same reference as `outer_allocator()`.

20.7.7.3 `scoped_allocator_adaptor` globals [allocator.adaptor.globals]

```
template<Allocator OuterA1, Allocator OuterA2, Allocator InnerA>
  bool operator==(const scoped_allocator_adaptor<OuterA1, InnerA>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerA>& b);
```

1 *Returns:* `a.outer_allocator() == b.outer_allocator()`
`&& a.inner_allocator() == b.inner_allocator()`

```
template<Allocator OuterA1, Allocator OuterA2, Allocator InnerA>
  bool operator!=(const scoped_allocator_adaptor<OuterA1, InnerA>& a,
                  const scoped_allocator_adaptor<OuterA2, InnerA>& b);
```

2 *Returns:* `!(a == b)`

20.7.8 Raw storage iterator [storage.iterator]

1 `raw_storage_iterator` is provided to enable algorithms to store their results into uninitialized memory. The formal template parameter `OutputIterator` is required to have its `operator*` return an object for which `operator&` is defined and returns a pointer to `T`, and is also required to satisfy the requirements of an output iterator (24.1.3).

```
namespace std {
  template <class OutputIterator, class T>
  class raw_storage_iterator
    : public iterator<output_iterator_tag,void,void,void,void> {
  public:
    explicit raw_storage_iterator(OutputIterator x);

    raw_storage_iterator<OutputIterator,T>& operator*();
    raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
    raw_storage_iterator<OutputIterator,T>& operator++();
    raw_storage_iterator<OutputIterator,T>  operator++(int);
  };
}
```

```
raw_storage_iterator(OutputIterator x);
```

2 *Effects:* Initializes the iterator to point to the same value to which `x` points.

```
raw_storage_iterator<OutputIterator,T>& operator*();
```

3 *Returns:* `*this`

```
raw_storage_iterator<OutputIterator,T>& operator=(const T& element);
```

4 *Effects:* Constructs a value from `element` at the location to which the iterator points.

5 *Returns:* A reference to the iterator.

```
raw_storage_iterator<OutputIterator,T>& operator++();
```

6 *Effects:* Pre-increment: advances the iterator and returns a reference to the updated iterator.

```
raw_storage_iterator<OutputIterator,T> operator++(int);
```

7 *Effects:* Post-increment: advances the iterator and returns the old value of the iterator.

20.7.9 Temporary buffers [temporary.buffer]

```
template <class T>
  pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
```

1 *Effects:* Obtains a pointer to storage sufficient to store up to `n` adjacent `T` objects. It is implementation-defined whether over-aligned types are supported (3.11).

2 *Returns:* A `pair` containing the buffer's address and capacity (in the units of `sizeof(T)`), or a pair of 0 values if no storage can be obtained or if `n <= 0`.

```
template <class T> void return_temporary_buffer(T* p);
```

3 *Effects:* Deallocates the buffer to which `p` points.

4 *Requires:* The buffer shall have been previously allocated by `get_temporary_buffer`.

20.7.10 `construct_element` [construct.element]

```
template <Allocator Alloc, class T, class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
    void construct_element(Alloc& a, T& r, Args&&... args);
```

1 [*Note:* The appropriate overload of the `construct_element` function is called from within containers to construct elements during insertion operations and to move elements during reallocation operations. It automates the process of determining whether the scoped allocator model is in use and transmitting the inner allocator for scoped allocators. — *end note*]

2 *Effects:* `AllocatableElement<Alloc, T, Args&&...>::construct_element( a, addressof(r), forward<Args>(args)...)`

20.7.11 Specialized algorithms [specialized.algorithms]

1 All the iterators that are used as formal template parameters in the following algorithms are required to have their `operator*` return an object for which `operator&` is defined and returns a pointer to `T`. In the algorithm `uninitialized_copy`, the formal template parameter `InputIterator` is required to satisfy the requirements of an input iterator (24.1.2). In all of the following algorithms, the formal template parameter `ForwardIterator` is required to satisfy the requirements of a forward iterator (24.1.4) and also to satisfy the requirements of a mutable iterator (24.1), and is required to have the property that no exceptions are thrown from increment, assignment, comparison, or dereference of valid iterators. In the following algorithms, if an exception is thrown there are no effects.

20.7.11.1 `addressof` [object.addressof]

```
template <ObjectType T> T* addressof(T& r);
template <ObjectType T> T* addressof(T&& r);
```

Returns: the actual address of the object referenced by `r`, even in the presence of an overloaded `operator&`.

20.7.11.2 `uninitialized_copy` [uninitialized.copy]

```
template <class InputIterator, class ForwardIterator>
  ForwardIterator uninitialized_copy(InputIterator first, InputIterator last,
                                     ForwardIterator result);
```

1 *Effects:*

```
for (; first != last; ++result, ++first)
  new (static_cast<void*>(&*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
```

2 *Returns:* `result`

```
template <class InputIterator, class Size, class ForwardIterator>
  ForwardIterator uninitialized_copy_n(InputIterator first, Size n,
                                       ForwardIterator result);
```

3 *Effects:*

```
for ( ; n > 0; ++result, ++first, --n) {
  new (static_cast<void*>(&*result))
    typename iterator_traits<ForwardIterator>::value_type(*first);
}
```

4 *Returns:* `result`

20.7.11.3 `uninitialized_fill` [uninitialized.fill]

```
template <class ForwardIterator, class T>
  void uninitialized_fill(ForwardIterator first, ForwardIterator last,
                          const T& x);
```

1 *Effects:*

```
for (; first != last; ++first)
  new (static_cast<void*>(&*first))
    typename iterator_traits<ForwardIterator>::value_type(x);
```

20.7.11.4 `uninitialized_fill_n` [uninitialized.fill.n]

```
template <class ForwardIterator, class Size, class T>
  void uninitialized_fill_n(ForwardIterator first, Size n, const T& x);
```

1 *Effects:*

```
for (; n--; ++first)
  new (static_cast<void*>(&*first))
    typename iterator_traits<ForwardIterator>::value_type(x);
```

20.7.12 Class template `unique_ptr` [unique.ptr]

1 Template `unique_ptr` stores a pointer to an object and deletes that object using the associated deleter when it is itself destroyed (such as when leaving block scope (6.7)).

2 The `unique_ptr` provides a semantics of strict ownership. A `unique_ptr` owns the object it holds a pointer to. A `unique_ptr` is not `CopyConstructible`, nor `CopyAssignable`, however it is `MoveConstructible` and `MoveAssignable`. The template parameter `T` of `unique_ptr` may be an incomplete type. [*Note:* The uses of `unique_ptr` include providing exception safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. — *end note*]

```
namespace std {
  template<class T> struct default_delete;
  template<class T> struct default_delete<T[]>;

  template<class T, class D = default_delete<T>> class unique_ptr;
  template<class T, class D> class unique_ptr<T[], D>;

  template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
  template<class T, class D> void swap(unique_ptr<T, D>&& x, unique_ptr<T, D>& y);
  template<class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>&& y);

  template<class T1, class D1, class T2, class D2>
    bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2>
    bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2>
    bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2>
    bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2>
    bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
  template<class T1, class D1, class T2, class D2>
    bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
}
```

20.7.12.1 Default deleters [unique.ptr.dltr]

20.7.12.1.1 `default_delete` [unique.ptr.dltr.dflt]

```
namespace std {
  template <class T> struct default_delete {
    default_delete();
    template <class U> default_delete(const default_delete<U>&);
    void operator()(T*) const;
  };
}
```

`default_delete();`

Effects: Default constructs a `default_delete`.

`template <class U> default_delete(const default_delete<U>& other);`

1 *Effects:* Constructs a `default_delete` from a `default_delete<U>`.

`void operator()(T *ptr) const;`

2 *Effects:* calls `delete` on `ptr`. A diagnostic is required if `T` is an incomplete type.

20.7.12.1.2 `default_delete<T[]>` [unique.ptr.dltr.dflt1]

```
namespace std {
  template <class T> struct default_delete<T[]> {
    void operator()(T*) const;
  };
}
```

```
void operator()(T* ptr) const;
```

1 `operator()` calls `delete[]` on `ptr`. A diagnostic is required if `T` is an incomplete type.

20.7.12.1.3 `default_delete<T[N]>` [unique.ptr.dltr.dflt2]

20.7.12.2 `unique_ptr` for single objects [unique.ptr.single]

```
namespace std {
  template <class T, class D = default_delete<T>> class unique_ptr {
  public:
    typedef implementation-defined pointer;
    typedef T element_type;
    typedef D deleter_type;

    // constructors
    unique_ptr();
    explicit unique_ptr(pointer p);
    unique_ptr(pointer p, implementation-defined d);
    unique_ptr(pointer p, implementation-defined d);
    unique_ptr(uniquePtr&& u);
    unique_ptr(nullptr_t) : unique_ptr() { }
    template <class U, class E> unique_ptr(unique_ptr<U, E>&& u);

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u);
    template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u);
    unique_ptr& operator=(unspecified-pointer-type);

    // observers
    typename add_lvalue_reference<T>::type operator*() const;
    pointer operator->() const;
    pointer get() const;
    deleter_type& get_deleter();
    const deleter_type& get_deleter() const;
    explicit operator bool() const;

    // modifiers
    pointer release();
    void reset(pointer p = pointer());
    void swap(unique_ptr&& u);

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    template <class U, class E> unique_ptr(const unique_ptr<U, E>&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
    template <class U, class E> unique_ptr& operator=(const unique_ptr<U, E>&) = delete;
  };
}
```

1 The default type for the template parameter `D` is `default_delete`. A client-supplied template argument `D` shall be a function pointer or functor for which, given a value `d` of type `D` and a pointer `ptr`

of type `T*`, the expression `d(ptr)` is valid and has the effect of deallocating the pointer as appropriate for that deleter. `D` may also be an lvalue-reference to a deleter.

2 If the deleter `D` maintains state, it is intended that this state stay with the associated pointer as ownership is transferred from `unique_ptr` to `unique_ptr`. The deleter state need never be copied, only moved or swapped as pointer ownership is moved around. That is, the deleter need only be `MoveConstructible`, `MoveAssignable`, and `Swappable`, and need not be `CopyConstructible` (unless copied into the `unique_ptr`) nor `CopyAssignable`.

3 If the type `remove_reference<D>::type::pointer` exists, then `unique_ptr<T, D>::pointer` shall be a synonym for `remove_reference<D>::type::pointer`. Otherwise `unique_ptr<T, D>::pointer` shall be a synonym for `T*`. The type `unique_ptr<T, D>::pointer` shall be CopyConstructible (Table 20.1.8) and CopyAssignable (Table 20.1.8).

20.7.12.2.1 `unique_ptr` constructors [unique.ptr.single.ctor]

```
unique_ptr();
```

1 *Requires:* `D` shall be default constructible, and that construction shall not throw an exception. `D` shall not be a reference type or pointer type (diagnostic required).

2 *Effects:* Constructs a `unique_ptr` which owns nothing.

3 *Postconditions:* `get() == 0`. `get_deleter()` returns a reference to a default constructed deleter `D`.

4 *Throws:* nothing.

```
unique_ptr(pointer p);
```

5 *Requires:* `D` shall be default constructible, and that construction shall not throw an exception.

6 *Effects:* Constructs a `unique_ptr` which owns `p`.

7 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to a default constructed deleter `D`.

8 *Throws:* nothing.

```
unique_ptr(pointer p, implementation-defined d);
unique_ptr(pointer p, implementation-defined d);
```

9 The signature of these constructors depends upon whether `D` is a reference type or not. If `D` is non-reference type `A`, then the signatures are:

```
unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, A&& d);
```

10 If `D` is an lvalue-reference type `A&`, then the signatures are:

```
unique_ptr(pointer p, A& d);
unique_ptr(pointer p, A&& d);
```

11 If `D` is an lvalue-reference type `const A&`, then the signatures are:

```
unique_ptr(pointer p, const A& d);
unique_ptr(pointer p, const A&& d);
```

12 *Requires:* If `D` is not an lvalue-reference type then

— If `d` is an lvalue or `const` rvalue then the first constructor of this pair will be selected. `D` must be `CopyConstructible` (Table 20.1.8), and this `unique_ptr` will hold a copy of `d`. The copy constructor of `D` shall not throw an exception.

— Otherwise `d` is a non-const rvalue and the second constructor of this pair will be selected. `D` need only be `MoveConstructible` (Table 20.1.8), and this `unique_ptr` will hold a value *move constructed* from `d`. The move constructor of `D` shall not throw an exception.

13 Otherwise `D` is an lvalue-reference type. `d` shall be reference-compatible with one of the constructors. If `d` is an rvalue, it will bind to the second constructor of this pair. That constructor shall emit a diagnostic. [*Note:* The diagnostic could be implemented using a `static_assert` which assures that `D` is not a reference type. — *end note*] Else `d` is an lvalue and will bind to the first constructor of this pair. The type which `D` references need not be `CopyConstructible` nor `MoveConstructible`. This `unique_ptr` will hold a `D` which refers to the lvalue `d`. [*Note:* `D` may not be an rvalue-reference type. — *end note*]

14 *Postconditions:* `get() == p`. `get_deleter()` returns a reference to the internally stored deleter. If `D` is a reference type then `get_deleter()` returns a reference to the lvalue `d`.

15 *Throws:* nothing.

[*Example:*

```
D d;
unique_ptr<int, D> p1(new int, D());         // D must be MoveConstructible
unique_ptr<int, D> p2(new int, d);           // D must be Copyconstructible
unique_ptr<int, D&> p3(new int, d);          // p3 holds a reference to d
unique_ptr<int, const D&> p4(new int, D()); // error: rvalue deleter object combined
                                             // with reference deleter type
```

— *end example*]

```
unique_ptr(unique_ptr&& u);
```

16 *Requires:* If the deleter is not a reference type, construction of the deleter `D` from an rvalue `D` shall not throw an exception.

17 *Effects:* Constructs a `unique_ptr` which owns the pointer which `u` owns (if any). If the deleter is not a reference type, it is move constructed from `u`'s deleter, otherwise the reference is copy constructed from `u`'s deleter. After the construction, `u` no longer owns a pointer. [*Note:* The deleter constructor can be implemented with `std::forward<D>`. — *end note*]

18 *Postconditions:* `get() ==` value `u.get()` had before the construction. `get_deleter()` returns a reference to the internally stored deleter which was constructed from `u.get_deleter()`. If `D` is a reference type then `get_deleter()` and `u.get_deleter()` both reference the same lvalue deleter.

19 *Throws:* nothing.

```
template <class U, class E> unique_ptr(unique_ptr<U, E>&& u);
```

20 *Requires:* If `D` is not a reference type, construction of the deleter `D` from an rvalue of type `E` shall be well formed and shall not throw an exception. If `D` is a reference type, then `E` shall be the same type as `D` (diagnostic required). `unique_ptr<U, E>::pointer` shall be implicitly convertible to `pointer`. [*Note:* These requirements imply that `T` and `U` are complete types. — *end note*]

21 *Effects:* Constructs a `unique_ptr` which owns the pointer which `u` owns (if any). If the deleter is not a reference type, it is move constructed from `u`'s deleter, otherwise the reference is copy

constructed from `u`'s deleter. After the construction, `u` no longer owns a pointer. [*Note:* The deleter constructor can be implemented with `std::forward<D>`. — *end note*]

22 *Postconditions:* `get() ==` value `u.get()` had before the construction, modulo any required offset adjustments resulting from the cast from `unique_ptr<U, E>::pointer` to `pointer`. `get_deleter()` returns a reference to the internally stored deleter which was constructed from `u.get_deleter()`.

23 *Throws:* nothing.

20.7.12.2.2 `unique_ptr` destructor [unique.ptr.single.dtor]

```
~unique_ptr();
```

1 *Requires:* The expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions. [*Note:* The use of `default_delete` requires `T` to be a complete type. — *end note*]

2 *Effects:* If `get() == 0` there are no effects. Otherwise `get_deleter()(get())`.

3 *Throws:* nothing.

20.7.12.2.3 `unique_ptr` assignment [unique.ptr.single.asgn]

```
unique_ptr& operator=(unique_ptr&& u);
```

1 *Requires:* Assignment of the deleter `D` from an rvalue `D` shall not throw an exception.

2 *Effects:* `reset(u.release())` followed by a move assignment from `u`'s deleter to this deleter.

3 *Postconditions:* This `unique_ptr` now owns the pointer which `u` owned, and `u` no longer owns it. [*Note:* If `D` is a reference type, then the referenced lvalue deleters are move assigned. — *end note*]

4 *Returns:* `*this`.

5 *Throws:* nothing.

```
template <class U, class E> unique_ptr& operator=(unique_ptr<U, E>&& u);
```

6 *Requires:* Assignment of the deleter `D` from an rvalue `D` shall not throw an exception. `unique_ptr<U, E>::pointer` shall be implicitly convertible to `pointer`. [*Note:* These requirements imply that `T` and `U` are complete types. — *end note*]

7 *Effects:* `reset(u.release())` followed by a move assignment from `u`'s deleter to this deleter. If either `D` or `E` is a reference type, then the referenced lvalue deleter participates in the move assignment.

8 *Postconditions:* This `unique_ptr` now owns the pointer which `u` owned, and `u` no longer owns it.

9 *Returns:* `*this`.

10 *Throws:* nothing.

```
unique_ptr& operator=(unspecified-pointer-type);
```

Assigns from the literal `0` or `NULL`. [*Note:* The *unspecified-pointer-type* is often implemented as a pointer to a private data member, avoiding many of the implicit conversion pitfalls. — *end note*]

11 *Effects:* `reset()`.

Postcondition: `get() == 0`

Returns: `*this`.

Throws: nothing.

20.7.12.2.4 `unique_ptr` observers [unique.ptr.single.observers]

```
typename add_lvalue_reference<T>::type operator*() const;
```

Requires: `get() != 0`.

Returns: `*get()`.

Throws: nothing.

```
pointer operator->() const;
```

Requires: `get() != 0`.

Returns: `get()`.

Throws: nothing.

Note: use typically requires that `T` be a complete type.

```
pointer get() const;
```

Returns: The stored pointer.

Throws: nothing.

```
deleter_type& get_deleter();
const deleter_type& get_deleter() const;
```

Returns: A reference to the stored deleter.

Throws: nothing.

```
explicit operator bool() const;
```

Returns: `get() != 0`.

Throws: nothing.

20.7.12.2.5 `unique_ptr` modifiers [unique.ptr.single.modifiers]

```
pointer release();
```

Postcondition: `get() == 0`.

Returns: The value `get()` had at the start of the call to `release`.

Throws: nothing.

```
void reset(pointer p = pointer());
```

Requires: The expression `get_deleter()(get())` shall be well formed, shall have well-defined behavior, and shall not throw exceptions.

Effects: If `get() == 0` there are no effects. Otherwise `get_deleter()(get())`.

Postconditions: `get() == p`.

7 *Throws:* nothing.

```
void swap(unique_ptr&& u);
```

8 *Requires:* The deleter `D` shall be `Swappable` and shall not throw an exception under `swap`.

9 *Effects:* The stored pointers of `this` and `u` are exchanged. The stored deleters are `swap`'d (unqualified).

10 *Throws:* nothing.

20.7.12.3 `unique_ptr` for array objects with a runtime length [unique.ptr.runtime]

```
namespace std {
  template <class T, class D> class unique_ptr<T[], D> {
  public:
    typedef implementation-defined pointer;
    typedef T element_type;
    typedef D deleter_type;

    // constructors
    unique_ptr();
    explicit unique_ptr(pointer p);
    unique_ptr(pointer p, implementation-defined d);
    unique_ptr(pointer p, implementation-defined d);
    unique_ptr(uniquePtr&& u);
    unique_ptr(nullptr_t) : unique_ptr() { }

    // destructor
    ~unique_ptr();

    // assignment
    unique_ptr& operator=(unique_ptr&& u);
    unique_ptr& operator=(unspecified-pointer-type);

    // observers
    T& operator[](size_t i) const;
    pointer get() const;
    deleter_type& get_deleter();
    const deleter_type& get_deleter() const;
    explicit operator bool() const;

    // modifiers
    pointer release();
    void reset(pointer p = pointer());
    void swap(unique_ptr&& u);

    // disable copy from lvalue
    unique_ptr(const unique_ptr&) = delete;
    unique_ptr& operator=(const unique_ptr&) = delete;
  };
}
```

1 A specialization for array types is provided with a slightly altered interface.

— Conversions among different types of `unique_ptr<T[], D>` or to or from the non-array forms of `unique_ptr` are disallowed (diagnostic required).

— Pointers to types derived from `T` are rejected by the constructors, and by `reset`.

— The observers `operator*` and `operator->` are not provided.

— The indexing observer `operator[]` is provided.

— The default deleter will call `delete[]`.

2 Descriptions are provided below only for member functions that have behavior different from the primary template.

3 The template argument `T` shall be a complete type.

20.7.12.3.1 `unique_ptr` constructors [unique.ptr.runtime.ctor]

```
unique_ptr(pointer p);
unique_ptr(pointer p, implementation-defined d);
unique_ptr(pointer p, implementation-defined d);
```

These constructors behave the same as in the primary template except that they do not accept pointer types which are convertible to `pointer`. [*Note:* One implementation technique is to create private templated overloads of these members. — *end note*]

20.7.12.3.2 `unique_ptr` observers [unique.ptr.runtime.observers]

```
T& operator[](size_t i) const;
```

1 *Requires:* `i <` the size of the array to which the stored pointer points.

2 *Returns:* `get()[i]`.

3 *Throws:* nothing.

20.7.12.3.3 `unique_ptr` modifiers [unique.ptr.runtime.modifiers]

```
void reset(pointer p = pointer());
```

1 *Requires:* Does not accept pointer types which are convertible to `pointer` (diagnostic required). [*Note:* One implementation technique is to create a private templated overload. — *end note*]

2 *Effects:* If `get() == 0` there are no effects. Otherwise `get_deleter()(get())`.

3 *Postcondition:* `get() == p`.

4 *Throws:* nothing.

20.7.12.4 `unique_ptr` specialized algorithms [unique.ptr.special]

```
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
template <class T, class D> void swap(unique_ptr<T, D>&& x, unique_ptr<T, D>& y);
template <class T, class D> void swap(unique_ptr<T, D>& x, unique_ptr<T, D>& y);
```

1 *Effects:* Calls `x.swap(y)`.

```
template <class T1, class D1, class T2, class D2>
  bool operator==(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

2 *Returns:* `x.get() == y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator!=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

3 *Returns:* `x.get() != y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator<(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

4 *Returns:* `x.get() < y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator<=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

5 *Returns:* `x.get() <= y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator>(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

6 *Returns:* `x.get() > y.get()`.

```
template <class T1, class D1, class T2, class D2>
  bool operator>=(const unique_ptr<T1, D1>& x, const unique_ptr<T2, D2>& y);
```

7 *Returns:* `x.get() >= y.get()`.

20.7.13 Smart pointers [util.smartptr]

20.7.13.1 Class `bad_weak_ptr` [util.smartptr.weakptr]

```
namespace std {
  class bad_weak_ptr: public std::exception {
  public:
    bad_weak_ptr();
  };
} // namespace std
```

1 An exception of type `bad_weak_ptr` is thrown by the `shared_ptr` constructor taking a `weak_ptr`.

```
bad_weak_ptr();
```

2 *Postconditions:* `what()` returns `"bad_weak_ptr"`.

3 *Throws:* nothing.

20.7.13.2 Class template `shared_ptr` [util.smartptr.shared]

1 The `shared_ptr` class template stores a pointer, usually obtained via `new`. `shared_ptr` implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer. A `shared_ptr` object is *empty* if it does not own a pointer.

```
namespace std {
  template<class T> class shared_ptr {
  public:
    typedef T element_type;

    // 20.7.13.2.1, constructors:
    shared_ptr();
```

```
  template<class Y> explicit shared_ptr(Y* p);
  template<class Y, class D> shared_ptr(Y* p, D d);
  template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
  template<class Y> shared_ptr(const shared_ptr<Y>& r, T *p);
  shared_ptr(const shared_ptr& r);
  template<class Y> shared_ptr(const shared_ptr<Y>& r);
  shared_ptr(shared_ptr&& r);
  template<class Y> shared_ptr(shared_ptr<Y>&& r);
  template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
  template<class Y> explicit shared_ptr(auto_ptr<Y>&& r);
  template <class Y, class D> explicit shared_ptr(const unique_ptr<Y, D>& r) = delete;
  template <class Y, class D> explicit shared_ptr(unique_ptr<Y, D>&&) r};
  shared_ptr(nullptr_t) : shared_ptr() { }

  // 20.7.13.2.2, destructor:
  ~shared_ptr();

  // 20.7.13.2.3, assignment:
  shared_ptr& operator=(const shared_ptr& r);
  template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r);
  shared_ptr& operator=(shared_ptr&& r);
  template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);
  template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
  template <class Y, class D> shared_ptr& operator=(const unique_ptr<Y, D>& r) = delete;
  template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

  // 20.7.13.2.4, modifiers:
  void swap(shared_ptr&& r);
  void reset();
  template<class Y> void reset(Y* p);
  template<class Y, class D> void reset(Y* p, D d);
  template<class Y, class D, class A> void reset(Y* p, D d, A a);

  // 20.7.13.2.5, observers:
  T* get() const;
  T& operator*() const;
  T* operator->() const;
  long use_count() const;
  bool unique() const;
  explicit operator bool() const;
  template <class U> bool owner_before(shared_ptr<U> const& b) const;
  template <class U> bool owner_before(weak_ptr<U> const& b) const;
};

// 20.7.13.2.6, shared_ptr creation
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// 20.7.13.2.7, shared_ptr comparisons:
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b);
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b);
template<class T, class U>
```

```
    bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b);

  // 20.7.13.2.8, shared_ptr I/O:
  template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);

  // 20.7.13.2.9, shared_ptr specialized algorithms:
  template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);
  template<class T> void swap(shared_ptr<T>&& a, shared_ptr<T>& b);
  template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>&& b);

  // 20.7.13.2.10, shared_ptr casts:
  template<class T, class U>
    shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
  template<class T, class U>
    shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r);
  template<class T, class U>
    shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r);

  // 20.7.13.2.11, shared_ptr get_deleter:
  template<class D, class T> D* get_deleter(const shared_ptr<T>& p);
} // namespace std
```

2 Specializations of `shared_ptr` shall be CopyConstructible, Assignable, and LessThanComparable, allowing their use in standard containers. Specializations of `shared_ptr` shall be convertible to `bool`, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

3 [*Example:*

```
if(shared_ptr<X> px = dynamic_pointer_cast<X>(py)) {
  // do something with px
}
```

— *end example*]

20.7.13.2.1 `shared_ptr` constructors [util.smartptr.shared.const]

```
shared_ptr();
```

1 *Effects:* Constructs an *empty* `shared_ptr` object.

2 *Postconditions:* `use_count() == 0 && get() == 0`.

3 *Throws:* nothing.

```
template<class Y> explicit shared_ptr(Y* p);
```

4 *Requires:* `p` shall be convertible to `T*`. `Y` shall be a complete type. The expression `delete p` shall be well formed, shall have well defined behavior, and shall not throw exceptions.

5 *Effects:* Constructs a `shared_ptr` object that *owns* the pointer `p`.

6 *Postconditions:* `use_count() == 1 && get() == p`.

7 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

8 *Exception safety:* If an exception is thrown, `delete p` is called.

```
template<class Y, class D> shared_ptr(Y* p, D d);
template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
```

9 *Requires:* `p` shall be convertible to `T*`. `D` shall be `CopyConstructible`. The copy constructor and destructor of `D` shall not throw exceptions. The expression `d(p)` shall be well formed, shall have well defined behavior, and shall not throw exceptions. `A` shall be an allocator (20.7.2.2). The copy constructor and destructor of `A` shall not throw exceptions.

10 *Effects:* Constructs a `shared_ptr` object that *owns* the pointer `p` and the deleter `d`. The second constructor shall use a copy of `a` to allocate memory for internal use.

11 *Postconditions:* `use_count() == 1 && get() == p`.

12 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

13 *Exception safety:* If an exception is thrown, `d(p)` is called.

```
template<class Y> shared_ptr(const shared_ptr<Y>& r, T *p);
```

14 *Effects:* Constructs a `shared_ptr` instance that stores `p` and *shares ownership* with `r`.

15 *Postconditions:* `get() == p && use_count() == r.use_count()`

16 *Throws:* nothing.

17 [*Note:* to avoid the possibility of a dangling pointer, the user of this constructor must ensure that `p` remains valid at least until the ownership group of `r` is destroyed. — *end note*]

18 [*Note:* this constructor allows creation of an *empty* `shared_ptr` instance with a non-NULL stored pointer. — *end note*]

```
shared_ptr(const shared_ptr& r);
template<class Y> shared_ptr(const shared_ptr<Y>& r);
```

19 *Requires:* The second constructor shall not participate in the overload resolution unless `Y*` is implicitly convertible to `T*`.

20 *Effects:* If `r` is *empty*, constructs an *empty* `shared_ptr` object; otherwise, constructs a `shared_ptr` object that *shares ownership* with `r`.

21 *Postconditions:* `get() == r.get() && use_count() == r.use_count()`.

22 *Throws:* nothing.

```
shared_ptr(shared_ptr&& r);
template<class Y> shared_ptr(shared_ptr<Y>&& r);
```

23 *Requires:* For the second constructor `Y*` shall be convertible to `T*`.

24 *Effects:* Move-constructs a `shared_ptr` instance from `r`.

25 *Postconditions:* `*this` shall contain the old value of `r`. `r` shall be *empty*. `r.get() == 0`.

26 *Throws:* nothing.

```
template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
```

27 *Requires:* `Y*` shall be convertible to `T*`.

28 *Effects:* Constructs a `shared_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

29 *Postconditions:* `use_count() == r.use_count()`.

30 *Throws:* `bad_weak_ptr` when `r.expired()`.

31 *Exception safety:* If an exception is thrown, the constructor has no effect.

```
template<class Y> explicit shared_ptr(auto_ptr<Y>&& r);
```

32 *Requires:* `r.release()` shall be convertible to `T*`. `Y` shall be a complete type. The expression `delete r.release()` shall be well formed, shall have well defined behavior, and shall not throw exceptions.

33 *Effects:* Constructs a `shared_ptr` object that stores and *owns* `r.release()`.

34 *Postconditions:* `use_count() == 1 && r.get() == 0`.

35 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

36 *Exception safety:* If an exception is thrown, the constructor has no effect.

```
template <class Y, class D> explicit shared_ptr(unique_ptr<Y, D>&&r);
```

37 *Effects:* Equivalent to `shared_ptr(r.release(), r.get_deleter())` when `D` is not a reference type, otherwise `shared_ptr(r.release(), ref(r.get_deleter()))`.

38 *Exception safety:* If an exception is thrown, the constructor has no effect.

20.7.13.2.2 `shared_ptr` destructor [util.smartptr.shared.dest]

```
~shared_ptr();
```

1 *Effects:*

— If `*this` is *empty* or shares ownership with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.

— Otherwise, if `*this` *owns* a pointer `p` and a deleter `d`, `d(p)` is called.

— Otherwise, `*this` *owns* a pointer `p`, and `delete p` is called.

2 *Throws:* nothing.

3 [*Note:* Since the destruction of `*this` decreases the number of instances that share ownership with `*this` by one, after `*this` has been destroyed all `shared_ptr` instances that shared ownership with `*this` will report a `use_count()` that is one less than its previous value. — *end note*]

20.7.13.2.3 `shared_ptr` assignment [util.smartptr.shared.assign]

```
shared_ptr& operator=(const shared_ptr& r);
template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
```

1 *Effects:* Equivalent to `shared_ptr(r).swap(*this)`.

2 *Returns:* `*this`.

3 [*Note:* The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments may be no-ops. — *end note*]

```
shared_ptr& operator=(shared_ptr&& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r);
```

4 *Effects:* Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

5 *Returns:* `*this`.

```
template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);
```

6 *Effects:* Equivalent to `shared_ptr(std::move(r)).swap(*this)`.

7 *Returns:* `*this`

20.7.13.2.4 `shared_ptr` modifiers [util.smartptr.shared.mod]

```
void swap(shared_ptr&& r);
```

1 *Effects:* Exchanges the contents of `*this` and `r`.

2 *Throws:* nothing.

```
void reset();
```

3 *Effects:* Equivalent to `shared_ptr().swap(*this)`.

```
template<class Y> void reset(Y* p);
```

4 *Effects:* Equivalent to `shared_ptr(p).swap(*this)`.

```
template<class Y, class D> void reset(Y* p, D d);
```

5 *Effects:* Equivalent to `shared_ptr(p, d).swap(*this)`.

```
template<class Y, class D, class A> void reset(Y* p, D d, A a);
```

6 *Effects:* Equivalent to `shared_ptr(p, d, a).swap(*this)`.

20.7.13.2.5 `shared_ptr` observers [util.smartptr.shared.obs]

```
T* get() const;
```

1 *Returns:* the stored pointer. Returns a null pointer if `*this` is empty.

2 *Throws:* nothing.

```
T& operator*() const;
```

3 *Requires:* `get() != 0`.

4 *Returns:* `*get()`.

5 *Throws:* nothing.

6 *Remarks:* When `T` is `void`, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

```
T* operator->() const;
```

7 *Requires:* `get() != 0`.

8 *Returns:* `get()`.

9 *Throws:* nothing.

```
long use_count() const;
```

10 *Returns:* the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or `0` when `*this` is *empty*.

11 *Throws:* nothing.

12 [*Note:* `use_count()` is not necessarily efficient. — *end note*]

```
bool unique() const;
```

13 *Returns:* `use_count() == 1`.

14 *Throws:* nothing.

15 [*Note:* `unique()` may be faster than `use_count()`. If you are using `unique()` to implement copy on write, do not rely on a specific value when `get() == 0`. — *end note*]

```
explicit operator bool() const;
```

16 *Returns:* `get() != 0`.

17 *Throws:* nothing.

```
template <class U> bool owner_before(shared_ptr<U> const& b) const;
template <class U> bool owner_before(weak_ptr<U> const& b) const;
```

18 *Returns:* an unspecified value such that

— `x.owner_before(y)` defines a strict weak ordering (25.3);

— under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

20.7.13.2.6 `shared_ptr` creation [util.smartptr.shared.create]

```
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);
```

1 *Requires:* The expression `new (pv) T(std::forward<Args>(args)...)`, where `pv` has type `void*` and points to storage suitable to hold an object of type `T`, shall be well formed. `A` shall be an *allocator* (20.7.2.2). The copy constructor and destructor of `A` shall not throw exceptions.

2 *Effects:* Allocates memory suitable for an object of type `T` and constructs an object in that memory via the placement new expression `new (pv) T()` or `new (pv) T(std::forward<Args>(args)...)`.

The template `allocate_shared` uses a copy of `a` to allocate memory. If an exception is thrown, the functions have no effect.

3 *Returns:* A `shared_ptr` instance that stores and owns the address of the newly constructed object of type `T`.

4 *Postconditions:* `get() != 0 && use_count() == 1`

5 *Throws:* `bad_alloc`, or an exception thrown from `A::allocate` or from the constructor of `T`.

6 *Remarks:* Implementations are encouraged, but not required, to perform no more than one memory allocation. [*Note:* this provides efficiency equivalent to an intrusive smart pointer. — *end note*]

7 [*Note:* these functions will typically allocate more memory than `sizeof(T)` to allow for internal bookkeeping structures such as the reference counts. — *end note*]

20.7.13.2.7 `shared_ptr` comparison [util.smartptr.shared.cmp]

```
template<class T, class U> bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b);
```

1 *Returns:* `a.get() == b.get()`.

2 *Throws:* nothing.

```
template<class T, class U> bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b);
```

3 *Returns:* `a.get() < b.get()`

4 For templates `greater`, `less`, `greater_equal`, and `less_equal`, the partial specializations for `shared_ptr` yield a total order, even if the built-in operators `<`, `>`, `<=`, `>=` do not. Moreover, `less<shared_ptr<T>>:: operator()(a, b)` shall return `std::less<T*>::operator()(a.get(), b.get())`.

5 *Throws:* nothing.

6 [*Note:* Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. — *end note*]

20.7.13.2.8 `shared_ptr` I/O [util.smartptr.shared.io]

```
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

1 *Effects:* `os << p.get();`.

2 *Returns:* `os`.

20.7.13.2.9 `shared_ptr` specialized algorithms [util.smartptr.shared.spec]

```
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);
template<class T> void swap(shared_ptr<T>&& a, shared_ptr<T>& b);
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>&& b);
```

1 *Effects:* Equivalent to `a.swap(b)`.

2 *Throws:* nothing.

20.7.13.2.10 shared_ptr casts [util.smartptr.shared.cast]

```
template<class T, class U> shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r);
```

1 *Requires:* The expression `static_cast<T*>(r.get())` shall be well formed.

2 *Returns:* If `r` is *empty*, an *empty* `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `static_cast<T*>(r.get())` and *shares ownership* with `r`.

3 *Postconditions:* `w.get() == static_cast<T*>(r.get())` and `w.use_count() == r.use_count()`, where `w` is the return value.

4 *Throws:* nothing.

5 [*Note:* The seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r);
```

6 *Requires:* The expression `dynamic_cast<T*>(r.get())` shall be well formed and shall have well defined behavior.

7 *Returns:*

- When `dynamic_cast<T*>(r.get())` returns a nonzero value, a `shared_ptr<T>` object that stores a copy of it and *shares ownership* with `r`;
- Otherwise, an *empty* `shared_ptr<T>` object.

8 *Postcondition:* `w.get() == dynamic_cast<T*>(r.get())`, where `w` is the return value.

9 *Throws:* nothing.

10 [*Note:* The seemingly equivalent expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

```
template<class T, class U> shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r);
```

11 *Requires:* The expression `const_cast<T*>(r.get())` shall be well formed.

12 *Returns:* If `r` is empty, an empty `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `const_cast<T*>(r.get())` and shares ownership with `r`.

13 *Postconditions:* `w.get() == const_cast<T*>(r.get())` and `w.use_count() == r.use_count()`, where `w` is the return value.

14 *Throws:* nothing.

15 [*Note:* The seemingly equivalent expression `shared_ptr<T>(const_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

20.7.13.2.11 get_deleter [util.smartptr.getdeleter]

```
template<class D, class T> D* get_deleter(const shared_ptr<T>& p);
```

1 *Returns:* If `p` *owns* a deleter `d` of type cv-unqualified `D`, returns `&d`; otherwise returns `0`. The returned pointer remains valid as long as there exists a `shared_ptr` instance that owns `d`. [*Note:* It is unspecified whether the pointer remains valid longer than that. This can happen if the implementation doesn't destroy the deleter until all `weak_ptr` instances that share ownership with `p` have been destroyed. — *end note*]

2 *Throws:* nothing.

20.7.13.3 Class template `weak_ptr` [util.smartptr.weak]

1 The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace std {
  template<class T> class weak_ptr {
  public:
    typedef T element_type;

    // constructors
    weak_ptr();
    template<class Y> weak_ptr(shared_ptr<Y> const& r);
    weak_ptr(weak_ptr const& r);
    template<class Y> weak_ptr(weak_ptr<Y> const& r);

    // destructor
    ~weak_ptr();

    // assignment
    weak_ptr& operator=(weak_ptr const& r);
    template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r);
    template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r);

    // modifiers
    void swap(weak_ptr& r);
    void reset();

    // observers
    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;
    template<class U> bool owner_before(const shared_ptr<U>& b);
    template<class U> bool owner_before(const weak_ptr<U>& b);

    // comparisons
    template <class Y> bool operator<(weak_ptr<Y> const&) const = delete;
    template <class Y> bool operator<=(weak_ptr<Y> const&) const = delete;
    template <class Y> bool operator>(weak_ptr<Y> const&) const = delete;
    template <class Y> bool operator>=(weak_ptr<Y> const&) const = delete;
  };

  // specialized algorithms
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);
} // namespace std
```

2 Specializations of `weak_ptr` shall be CopyConstructible, Assignable, and LessThanComparable, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

20.7.13.3.1 `weak_ptr` constructors [util.smartptr.weak.const]

```
weak_ptr();
```

1 *Effects:* Constructs an *empty* `weak_ptr` object.

2 *Postconditions:* `use_count() == 0`.

3 *Throws:* nothing.

```
weak_ptr(const weak_ptr& r);
template<class Y> weak_ptr(const weak_ptr<Y>& r);
template<class Y> weak_ptr(const shared_ptr<Y>& r);
```

4 *Requires:* The second and third constructors shall not participate in the overload resolution unless `Y*` is implicitly convertible to `T*`.

5 *Effects:* If `r` is *empty*, constructs an *empty* `weak_ptr` object; otherwise, constructs a `weak_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

6 *Postconditions:* `use_count() == r.use_count()`.

7 *Throws:* nothing.

20.7.13.3.2 `weak_ptr` destructor [util.smartptr.weak.dest]

```
~weak_ptr();
```

1 *Effects:* Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.

2 *Throws:* nothing.

20.7.13.3.3 `weak_ptr` assignment [util.smartptr.weak.assign]

```
weak_ptr& operator=(const weak_ptr& r);
template<class Y> weak_ptr& operator=(const weak_ptr<Y>& r);
template<class Y> weak_ptr& operator=(const shared_ptr<Y>& r);
```

1 *Effects:* Equivalent to `weak_ptr(r).swap(*this)`.

2 *Throws:* nothing.

3 *Remarks:* The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary.

20.7.13.3.4 `weak_ptr` modifiers [util.smartptr.weak.mod]

```
void swap(weak_ptr& r);
```

1 *Effects:* Exchanges the contents of `*this` and `r`.

2 *Throws:* nothing.

```
void reset();
```

3 *Effects:* Equivalent to `weak_ptr().swap(*this)`.

20.7.13.3.5 `weak_ptr` observers [util.smartptr.weak.obs]

```
long use_count() const;
```

1 *Returns:* `0` if `*this` is *empty*; otherwise, the number of `shared_ptr` instances that *share ownership* with `*this`.

2 *Throws:* nothing.

3 [*Note:* `use_count()` is not necessarily efficient. — *end note*]

```
bool expired() const;
```

4 *Returns:* `use_count() == 0`.

5 *Throws:* nothing.

6 [*Note:* `expired()` may be faster than `use_count()`. — *end note*]

```
shared_ptr<T> lock() const;
```

7 *Returns:* `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`.

8 *Throws:* nothing.

```
template<class U> bool owner_before(const shared_ptr<U>& b);
template<class U> bool owner_before(const weak_ptr<U>& b);
```

9 *Returns:* an unspecified value such that

— `x.owner_before(y)` defines a strict weak ordering as described in 25.3;

— under the equivalence relation defined by `owner_before`, `!a.owner_before(b) && !b.owner_before(a)`, two `shared_ptr` or `weak_ptr` instances are equivalent if and only if they share ownership or are both empty.

20.7.13.3.6 `weak_ptr` specialized algorithms [util.smartptr.weak.spec]

```
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b)
```

1 *Effects:* Equivalent to `a.swap(b)`.

2 *Throws:* nothing.

20.7.13.4 Class template `owner_less` [util.smartptr.ownerless]

1 The `owner_less` class template allows ownership-based mixed comparisons of shared and weak pointers.

```
namespace std {
  template <class T> struct owner_less;
  template <class T> struct owner_less<shared_ptr<T> >
    : binary_function<shared_ptr<T>, shared_ptr<T>, bool> {
    typedef bool result_type;
    bool operator()(shared_ptr<T> const&, shared_ptr<T> const&) const;
    bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
  };
  template <class T> struct owner_less<weak_ptr<T> >
    : binary_function<weak_ptr<T>, weak_ptr<T>, bool> {
    typedef bool result_type;
    bool operator()(weak_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(shared_ptr<T> const&, weak_ptr<T> const&) const;
    bool operator()(weak_ptr<T> const&, shared_ptr<T> const&) const;
  };
}
```

2 `operator()(x,y)` shall return `x.before(y)`. [*Note:*

— `operator()` defines a strict weak ordering as described in 25.3;

— under the equivalence relation defined by `operator()`, `!operator()(a,b) && !operator()(b,a)`, two `shared_ptr` or `weak_ptr` objects are equivalent if and only if they share ownership or are both empty.

— *end note*]

20.7.13.5 Class template `enable_shared_from_this` [util.smartptr.enab]

1 A class `T` can inherit from `enable_shared_from_this<T>` to inherit the `shared_from_this` member functions that obtain a *shared_ptr* instance pointing to `*this`.

2 [*Example:*

```
struct X: public enable_shared_from_this<X> {
};

int main() {
  shared_ptr<X> p(new X);
  shared_ptr<X> q = p->shared_from_this();
  assert(p == q);
  assert(!(p < q ) && !(q < p)); // p and q share ownership
}
```

— *end example*]

```
namespace std {
  template<class T> class enable_shared_from_this {
  protected:
    enable_shared_from_this();
    enable_shared_from_this(enable_shared_from_this const&);
    enable_shared_from_this& operator=(enable_shared_from_this const&);
    ~enable_shared_from_this();
  public:
    shared_ptr<T> shared_from_this();
    shared_ptr<T const> shared_from_this() const;
  };
} // namespace std
```

3 The template parameter `T` of `enable_shared_from_this` may be an incomplete type.

```
enable_shared_from_this();
enable_shared_from_this(const enable_shared_from_this<T>&);
```

4 *Effects:* Constructs an `enable_shared_from_this<T>` object.

5 *Throws:* nothing.

```
enable_shared_from_this<T>& operator=(const enable_shared_from_this<T>&);
```

6 *Returns:* `*this`.

7 *Throws:* nothing.

```
~enable_shared_from_this();
```

8 *Effects:* Destroys `*this`.

9 *Throws:* nothing.

```
shared_ptr<T>       shared_from_this();
shared_ptr<T const> shared_from_this() const;
```

10 *Requires:* `enable_shared_from_this<T>` shall be an accessible base class of `T`. `*this` shall be a subobject of an object `t` of type `T`. There shall be at least one `shared_ptr` instance `p` that *owns* `&t`.

11 *Returns:* A `shared_ptr<T>` object `r` that *shares ownership with* `p`.

12 *Postconditions:* `r.get() == this`.

13 [*Note:* a possible implementation is shown below:

```
template<class T> class enable_shared_from_this {
private:
  weak_ptr<T> __weak_this;
protected:
  enable_shared_from_this() {}
  enable_shared_from_this(enable_shared_from_this const &) {}
  enable_shared_from_this& operator=(enable_shared_from_this const &) { return *this; }
  ~enable_shared_from_this() {}
public:
  shared_ptr<T> shared_from_this() { return shared_ptr<T>(__weak_this); }
  shared_ptr<T const> shared_from_this() const { return shared_ptr<T const>(__weak_this); }
};
```

14 The `shared_ptr` constructors that create unique pointers can detect the presence of an `enable_shared_from_this` base and assign the newly created `shared_ptr` to its `__weak_this` member. — *end note*]

20.7.13.6 `shared_ptr` atomic access [util.smartptr.shared.atomic]

1 Concurrent access to a `shared_ptr` object from multiple threads does not introduce a data race if the access is done exclusively via the functions in this section and the instance is passed as their first argument.

2 The meaning of the arguments of type `memory_order` is explained in 29.1.

```
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);
```

3 *Returns:* `true` if atomic access to `*p` is lock-free, `false` otherwise.

4 *Throws:* nothing.

```
template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
```

5 *Returns:* `atomic_load_explicit(p, memory_order_seq_cst)`.

```
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
```

6 *Requires:* `mo` shall not be `memory_order_release` or `memory_order_acq_rel`.

7 *Returns:* `*p`.

8 *Throws:* nothing.

```
template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
```

9 *Effects:* `atomic_store_explicit(p, r, memory_order_seq_cst)`.

```
template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
```

10 *Requires:* `mo` shall not be `memory_order_acquire` or `memory_order_acq_rel`.

11 *Effects:* `p->swap(r)`.

12 *Throws:* nothing.

```
template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
```

13 *Returns:* `atomic_exchange_explicit(p, r, memory_order_seq_cst)`.

```
template<class T>
  shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                         memory_order mo);
```

14 *Effects:* `p->swap(r)`.

15 *Returns:* the previous value of `*p`.

16 *Throws:* nothing.

```
template<class T>
  bool atomic_compare_exchange_weak(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```

17 *Returns:* `atomic_compare_exchange_weak_explicit(p, v, w, memory_order_seq_cst, memory_order_seq_cst)`.

```
template<class T>
  bool atomic_compare_exchange_strong(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
```

18 *Returns:* `atomic_compare_exchange_strong_explicit(p, v, w, memory_order_seq_cst, memory_order_seq_cst)`.

```
template<class T>
  bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
template<class T>
  bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
```

19 *Requires:* `failure` shall not be `memory_order_release`, `memory_order_acq_rel`, or stronger than `success`.

20 *Effects:* If `*p` is equivalent to `*v`, assigns `w` to `*p` and has synchronization semantics corresponding to the value of `success`, otherwise assigns `*p` to `*v` and has synchronization semantics corresponding to the value of `failure`.

21 *Returns:* `true` if `*p` was equivalent to `*v`, `false` otherwise.

22 *Throws:* nothing.

23 *Remarks:* two `shared_ptr` objects are equivalent if they store the same pointer value and share ownership.

24 *Remarks:* the weak forms may fail spuriously. See 29.4.

20.7.13.7 Pointer safety [util.dynamic.safety]

1 A complete object is *declared reachable* while the number of calls to `declare_reachable` with an argument referencing the object exceeds the number of calls to `undeclare_reachable` with an argument referencing the object.

```
void declare_reachable(void *p);
```

2 *Requires:* `p` shall be a safely-derived pointer (3.7.4.3) or a null pointer value.

3 *Effects:* If `p` is not null, the complete object referenced by `p` is subsequently declared reachable (3.7.4.3).

4 *Throws:* May throw `std::bad_alloc` if the system cannot allocate additional memory that may be required to track objects declared reachable.

```
template <class T> T *undeclare_reachable(T *p);
```

5 *Requires:* If `p` is not null, the complete object referenced by `p` shall have been previously declared reachable, and shall be live (3.8) from the time of the call until the last `undeclare_reachable(p)` call on the object.

6 *Returns:* a safely derived copy of `p` which shall compare equal to `p`.

7 *Effects:* After a call to `undeclare_reachable(p)`, if `p` is not null and the complete object `q` referenced by `p` is no longer declared reachable, then dereferencing any pointer to `q` that is not safely derived results in undefined behavior. [*Note:* Since the returned pointer is safely derived, it may be used to access the referenced object, even if previously no safely derived pointer existed. — *end note*]

8 *Throws:* nothing.

9 [*Note:* It is expected that calls to `declare_reachable(p)` will consume a small amount of memory, in addition to that occupied by the referenced object, until the matching call to `undeclare_reachable(p)` is encountered. Long running programs should arrange that calls for short-lived objects are matched. — *end note*]

```
void declare_no_pointers(char *p, size_t n);
```

10 *Requires:* No bytes in the specified range have been previously registered with `declare_no_pointers()`. If the specified range is in an allocated object, then it must be entirely within a single allocated object. The object must be live until the corresponding `undeclare_no_pointers()` call. [*Note:* In a garbage-collecting implementation, the fact that a region in an object is registered with `declare_no_pointers()` should not prevent the object from being collected. — *end note*]

11 *Effects:* The `n` bytes starting at `p` no longer contain traceable pointer locations, independent of their type. Hence pointers located there may not be dereferenced if the object they point to was created by global `operator new` and not previously declared reachable. [*Note:* This may be used to inform a garbage collector or leak detector that this region of memory need not be traced. — *end note*]

12 *Throws:* nothing. [*Note:* Under some conditions implementations may need to allocate memory. However, the request can be ignored if memory allocation fails. — *end note*]

```
void undeclare_no_pointers(char *p, size_t n);
```

13 *Requires:* The same range must previously have been passed to `declare_no_pointers()`.

14 *Effects:* Unregisters a range registered with `declare_no_pointers()` for destruction. It must be called before the lifetime of the object ends.

15 *Throws:* nothing.

```
pointer_safety get_pointer_safety();
```

16 *Returns:* an enumeration value indicating the implementation's treatment of pointers that are not safely derived (3.7.4.3). Returns `pointer_safety::relaxed` if pointers that are not safely derived will be treated the same as pointers that are safely derived for the duration of the program. Returns `pointer_safety::preferred` if pointers that are not safely derived will be treated the same as pointers that are safely derived for the duration of the program but allows the implementation to hint that it could be desirable to avoid dereferencing pointers that are not safely derived as described. [*Example:* `pointer_safety::preferred` might be returned to detect if a leak detector is running to avoid spurious leak reports. — *end note*] Returns `pointer_safety::strict` if pointers that are not safely derived might be treated differently than pointers that are safely derived.

20.7.14 Align [ptr.align]

```
void *align(std::size_t alignment, std::size_t size,
    void *&ptr, std::size_t& space);
```

1 *Effects:* If it is possible to fit `size` bytes of storage aligned by `alignment` into the buffer pointed to by `ptr` with length `space`, the function updates `ptr` to point to the first possible address of such storage and decreases `space` by the number of bytes used for alignment. Otherwise, the function does nothing.

2 *Requires:*

— `alignment` shall be a fundamental alignment value or an extended alignment value supported by the implementation in this context

— `ptr` shall point to contiguous storage of at least `space` bytes

3 *Returns:* a null pointer if the requested aligned buffer would not fit into the available space, otherwise the adjusted value of `ptr`.

4 [*Note:* the function updates its `ptr` and `space` arguments so that it can be called repeatedly with possibly different `alignment` and `size` arguments for the same buffer.

20.7.15 C Library [c.malloc]

1 Table 43 describes the header `<cstdlib>`.

Table 43 — Header `<cstdlib>` synopsis

Type	Name(s)	
Functions:	`calloc`	`malloc`
	`free`	`realloc`

2 The contents are the same as the Standard C library header `<stdlib.h>`, with the following changes:

3 The functions `calloc()`, `malloc()`, and `realloc()` do not attempt to allocate storage by calling `::operator new()` (18.5).

4 The function `free()` does not attempt to deallocate storage by calling `::operator delete()`.

See also: ISO C Clause 7.11.2.

5 Storage allocated directly with `malloc()`, `calloc()`, or `realloc()` is implicitly declared reachable (see 3.7.4.3) on allocation, ceases to be declared reachable on deallocation, and need not cease to be declared reachable as the result of an `undeclare_reachable()` call. [*Note:* This allows existing C libraries to remain unaffected by restrictions on pointers that are not safely derived, at the expense of providing far fewer garbage collection and leak detection options for `malloc()`-allocated objects. It also allows `malloc()` to be implemented with a separate allocation arena, bypassing the normal `declare_reachable()` implementation. The above functions should never intentionally be used as a replacement for `declare_reachable()`, and newly written code is strongly encouraged to treat memory allocated with these functions as though it were allocated with `operator new`. — *end note*]

6 Table 44 describes the header `<cstring>`.

Table 44 — Header `<cstring>` synopsis

Type	Name(s)	
Macro:	`NULL`	
Type:	`size_t`	
Functions:	`memchr`	`memcmp`
`memcpy`	`memmove`	`memset`

7 The contents are the same as the Standard C library header `<string.h>`, with the change to `memchr()` specified in 21.5.

See also: ISO C Clause 7.11.2.

20.8 Time utilities [time]

1 This subclause describes the chrono library that provides generally useful time utilities.

Header `<chrono>` synopsis

```
namespace std {
namespace chrono {

template <class Rep, class Period = ratio<1>> class duration;
template <class Clock, class Duration = typename Clock::duration> class time_point;

}  // namespace chrono

// common_type traits
template <class Rep1, class Period1, class Rep2, class Period2>
  struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>>;

template <class Clock, class Duration1, class Duration2>
  struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>>;

namespace chrono {

// customization traits
```

```
template <class Rep> struct treat_as_floating_point;
template <class Rep> struct duration_values;

// duration arithmetic
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
  operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>>::type
  operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  operator*(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  operator*(const Rep1& s, const duration<Rep2, Period>& d);
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  operator/(const duration<Rep1, Period>& d, const Rep2& s);
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<Rep1, Rep2>::type
  operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// duration comparisons
template <class Rep1, class Period1, class Rep2, class Period2>
   bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
   bool operator!=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
   bool operator< (const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
   bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
   bool operator> (const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Rep2, class Period2>
   bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);

// duration_cast
template <class ToDuration, class Rep, class Period>
  ToDuration duration_cast(const duration<Rep, Period>& d);

// convenience typedefs
typedef duration<signed integral type of at least 64 bits,        nano> nanoseconds;
typedef duration<signed integral type of at least 55 bits,       micro> microseconds;
typedef duration<signed integral type of at least 45 bits,       milli> milliseconds;
typedef duration<signed integral type of at least 35 bits            > seconds;
typedef duration<signed integral type of at least 29 bits, ratio<  60>> minutes;
typedef duration<signed integral type of at least 23 bits, ratio<3600>> hours;

// time_point arithmetic
template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
  operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Rep1, class Period1, class Clock, class Duration2>
  time_point<Clock, typename common_type<duration<Rep1, Period1>, Duration2>::type>
```

```
  operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>>::type>
  operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
template <class Clock, class Duration1, class Duration2>
  typename common_type<Duration1, Duration2>::type
  operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// time_point comparisons
template <class Clock, class Duration1, class Duration2>
   bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   bool operator!=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   bool operator< (const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   bool operator<=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   bool operator> (const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
template <class Clock, class Duration1, class Duration2>
   bool operator>=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);

// time_point_cast

template <class ToDuration, class Clock, class Duration>
  time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);

// Clocks
class system_clock;
class monotonic_clock;
class high_resolution_clock;

}  // namespace chrono
}  // namespace std
```

20.8.1 Clock requirements [time.clock.req]

1 A clock is a bundle consisting of a native `duration`, a native `time_point`, and a function `now()` to get the current time_point. A clock shall meet the requirements in Table 45.

2 In Table 45 `C1` and `C2` denote clock types. `t1` and `t2` are values returned by `C1::now()` where the call returning `t1` happens before (1.10) the call returning `t2` and both of these calls happen before `C1::time_point::max()`.

Table 45 — Clock requirements

Expression	Return type	Operational semantics
`C1::rep`	An arithmetic type or a class emulating an arithmetic type	The representation type of the native `duration` and `time_point`.
`C1::period`	`ratio`	The tick period of the clock in seconds.
`C1::duration`	`chrono::duration<C1::rep, C1::period>`	The native `duration` type of the clock.

Table 45 — Clock requirements (continued)

Expression	Return type	Operational semantics
`C1::time_point`	`chrono::time_point<C1>` or `chrono::time_point<C2, C1::duration>`	The native `time_point` type of the clock. Different clocks may share a `time_point` definition if it is valid to compare their `time_point`s by comparing their respective `duration`s. `C1` and `C2` shall refer to the same epoch.
`C1::is_monotonic`	`const bool`	`true` if `t1 <= t2` is always `true`, otherwise `false`. [*Note:* A clock that can be adjusted backwards is not monotonic. — *end note*]
`C1::now()`	`C1::time_point`	Returns a `time_point` object representing the current point in time.

20.8.2 Time-related traits [time.traits]

20.8.2.1 `is_floating_point` [time.traits.is_fp]

```
template <class Rep> struct treat_as_floating_point
  : is_floating_point<Rep> { };
```

1 The `duration` template uses the `treat_as_floating_point` trait to help determine if a `duration` object can be converted to another `duration` with a different tick `period`. If `treat_as_floating_point<Rep>::value` is true, then `Rep` is a floating-point type and implicit conversions are allowed among `duration`s. Otherwise, the implicit convertibility depends on the tick `period`s of the `duration`s. If `Rep` is a class type which emulates a floating-point type, the author of `Rep` can specialize `treat_as_floating_point` so that `duration` will treat this `Rep` as if it were a floating-point type. Otherwise `Rep` is assumed to be an integral type or a class emulating an integral type.

20.8.2.2 `duration_values` [time.traits.duration_values]

```
template <class Rep>
struct duration_values {
public:
  static constexpr Rep zero();
  static constexpr Rep min();
  static constexpr Rep max();
};
```

1 The `duration` template uses the `duration_values` trait to construct special values of the durations representation (`Rep`). This is done because the representation might be a class type with behavior which requires some other implementation to return these special values. In that case, the author of that class type should specialize `duration_values` to return the indicated values.

```
static constexpr Rep zero();
```

2 *Returns:* `Rep(0)`. [*Note:* `Rep(0)` is specified instead of `Rep()` because `Rep()` may have some other meaning, such as an uninitialized value. — *end note*]

3 *Remark:* The value returned shall be the additive identity.

```
static constexpr Rep min();
```

4 *Returns:* `numeric_limits<Rep>::lowest()`.

5 *Remark:* The value returned shall compare less than or equal to `zero()`.

```
static constexpr Rep max();
```

6 *Returns:* `numeric_limits<Rep>::max()`.

7 *Remark:* The value returned shall compare greater than `zero()`.

20.8.2.3 Specializations of `common_type` [time.traits.specializations]

```
template <class Rep1, class Period1, class Rep2, class Period2>
struct common_type<chrono::duration<Rep1, Period1>, chrono::duration<Rep2, Period2>> {
  typedef chrono::duration<typename common_type<Rep1, Rep2>::type, see below}> type;
};
```

1 The `period` of the `duration` indicated by this specialization of `common_type` shall be the greatest common divisor of `Period1` and `Period2`. [*Note:* This can be computed by forming a ratio of the greatest common divisor of `Period1::num` and `Period2::num` and the least common multiple of `Period1::den` and `Period2::den`. — *end note*]

2 [*Note:* The `typedef` name `type` is a synonym for the `duration` with the largest tick `period` possible where both `duration` arguments will convert to it without requiring a division operation. The representation of this type is intended to be able to hold any value resulting from this conversion with no truncation eror, although floating-point durations may have round-off errors. — *end note*]

```
template <class Clock, class Duration1, class Duration2>
struct common_type<chrono::time_point<Clock, Duration1>, chrono::time_point<Clock, Duration2>> {
  typedef chrono::time_point<Clock, typename common_type<Duration1, Duration2>::type> type;
};
```

3 The common type of two `time_point` types is a `time_point` with the same clock as the two types and the common type of their two `duration`s.

20.8.3 Class template `duration` [time.duration]

1 A `duration` type measures time between two points in time (`time_point`s). A `duration` has a representation which holds a count of ticks and a tick period. The tick period is the amount of time which occurs from one tick to the next, in units of seconds. It is expressed as a rational constant using the template `ratio`.

```
template <class Rep, class Period = ratio<1>>
class duration {
public:
  typedef Rep    rep;
  typedef Period period;
private:
  rep rep_;  // exposition only
public:
  // 20.8.3.1, construct/copy/destroy:
  duration() = default;
```

```
  template <class Rep2>
      explicit duration(const Rep2& r);
  template <class Rep2, class Period2>
     duration(const duration<Rep2, Period2>& d);
  ~duration() = default;
  duration(const duration&) = default;
  duration& operator=(const duration&) = default;

  // 20.8.3.2, observer:
  rep count() const;

  // 20.8.3.3, arithmetic:
  duration  operator+() const;
  duration  operator-() const;
  duration& operator++();
  duration  operator++(int);
  duration& operator--();
  duration  operator--(int);

  duration& operator+=(const duration& d);
  duration& operator-=(const duration& d);

  duration& operator*=(const rep& rhs);
  duration& operator/=(const rep& rhs);

  // 20.8.3.4, special values:
  static constexpr duration zero();
  static constexpr duration min();
  static constexpr duration max();
};
```

2 *Requires:* `Rep` shall be an arithmetic type or a class emulating an arithmetic type. If a program instantiates `duration` with a `duration` type for the template argument `Rep` a diagnostic is required.

3 *Requires:* `Period` shall be a specialization of `ratio`, diagnostic required.

4 *Requires:* `Period::num` shall be positive, diagnostic required.

5 *Requires:* Members of `duration` shall not throw exceptions other than those thrown by the indicated operations on their representations.

[*Example:*

```
duration<long, ratio<60>> d0;       // holds a count of minutes using a long
duration<long long, milli> d1;      // holds a count of milliseconds using a long long
duration<double, ratio<1, 30>>  d2; // holds a count with a tick period of 1/30 of a second
                                    // (30 Hz) using a double
```

— *end example*]

20.8.3.1 `duration` constructors [time.duration.cons]

```
template <class Rep2>
  explicit duration(const Rep2& r);
```

1 *Requires:* `Rep2` shall be implicitly convertible to `rep` and

— `treat_as_floating_point<rep>::value` shall be `true` or

— `treat_as_floating_point<Rep2>::value` shall be `false`.

Diagnostic required. [*Example:*

```
duration<int, milli> d(3);          // OK
duration<int, milli> d(3.5);        // error
```

— *end example*]

2 *Effects:* Constructs an object of type `duration`.

3 *Postcondition:* `count() == static_cast<rep>(r)`.

```
template <class Rep2, class Period2>
  duration(const duration<Rep2, Period2>& d);
```

4 *Requires:* `treat_as_floating_point<rep>::value` shall be `true` or `ratio_divide<Period2, period>::type::den` shall be `1`. Diagnostic required. [*Note:* This requirement prevents implicit truncation error when converting between integral-based `duration` types. Such a construction could easily lead to confusion about the value of the `duration`. — *end note*] [*Example:*

```
duration<int, milli> ms(3);
duration<int, micro> us = ms;       // OK
duration<int, milli> ms2 = us;      // error
```

— *end example*]

5 *Effects:* Constructs an object of type `duration`, constructing `rep_` from `duration_cast<duration>(d).count()`.

20.8.3.2 `duration` observer [time.duration.observer]

```
rep count() const;
```

1 *Returns:* `rep_`.

20.8.3.3 `duration` arithmetic [time.duration.arithmetic]

```
duration operator+() const;
```

1 *Returns:* `*this`.

```
duration operator-() const;
```

2 *Returns:* `duration(-rep_);`.

```
duration& operator++();
```

3 *Effects:* `++rep_`.

4 *Returns:* `*this`.

```
duration operator++(int);
```

5 *Returns:* `duration(rep_++);`.

```
duration& operator-{-}();
```

Effects: `--rep_`.

Returns: `*this`.

```
duration operator-{-}(int);
```

Returns: `duration(rep_--);`.

```
duration& operator+=(const duration& d);
```

Effects: `rep_ += d.count()`.

Returns: `*this`.

```
duration& operator-=(const duration& d);
```

Effects: `rep_ -= d.count()`.

Returns: `*this`.

```
duration& operator*=(const rep& rhs);
```

Effects: `rep_ *= rhs`.

Returns: `*this`.

```
duration& operator/=(const rep& rhs);
```

Effects: `rep_ /= rhs`.

Returns: `*this`.

20.8.3.4 `duration` special values [time.duration.special]

```
static constexpr duration zero();
```

Returns: `duration(duration_values<rep>::zero())`.

```
static constexpr duration min();
```

Returns: `duration(duration_values<rep>::min())`.

```
static constexpr duration max();
```

Returns: `duration(duration_values<rep>::max())`.

20.8.3.5 `duration` non-member arithmetic [time.duration.nonmember]

In the function descriptions that follow, `CD` represents the return type of the function. `CR(A,B)` represents `common_type<A, B>::type`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>{>}::type
  operator+(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

Returns: `CD(lhs) += rhs`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<duration<Rep1, Period1>, duration<Rep2, Period2>{>}::type
  operator-(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

Returns: `CD(lhs) -= rhs`.

```
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  operator*(const duration<Rep1, Period>& d, const Rep2& s);
```

4 *Requires:* `Rep2` shall be implicitly convertible to CR(Rep1, Rep2). Diagnostic required.

5 *Returns:* `duration<CR(Rep1, Rep2), Period>(d) *= s`.

```
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  operator*(const Rep1& s, const duration<Rep2, Period>& d);
```

6 *Requires:* `Rep1` shall be implicitly convertible to CR(Rep1, Rep2). Diagnostic required.

7 *Returns:* `d * s`.

```
template <class Rep1, class Period, class Rep2>
  duration<typename common_type<Rep1, Rep2>::type, Period>
  operator/(const duration<Rep1, Period>& d, const Rep2& s);
```

8 *Requires:* `Rep2` shall be implicitly convertible to CR(Rep1, Rep2) and `Rep2` shall not be an instantiation of `duration`. Diagnostic required.

9 *Returns:* `duration<CR, Period>(d) /= s`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  typename common_type<Rep1, Rep2>::type
  operator/(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

10 *Returns:* `CD(lhs).count() / CD(rhs).count()`.

20.8.3.6 `duration` comparisons [time.duration.comparisons]

1 In the function descriptions that follow, `CT` represents `common_type<A, B>::type`, where `A` and `B` are the types of the two arguments to the function.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  bool operator==(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

2 *Returns:* `CT(lhs).count() == CT(rhs.count())`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  bool operator!=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

3 *Returns:* `!(lhs == rhs)`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  bool operator<(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

4 *Returns:* `CT(lhs).count() < CT(rhs).count()`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  bool operator<=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

5 *Returns:* `!(rhs < lhs)`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  bool operator>(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

6 *Returns:* `rhs < lhs`.

```
template <class Rep1, class Period1, class Rep2, class Period2>
  bool operator>=(const duration<Rep1, Period1>& lhs, const duration<Rep2, Period2>& rhs);
```

7 *Returns:* `!(lhs < rhs)`.

20.8.3.7 `duration_cast` [time.duration.cast]

```
template <class ToDuration, class Rep, class Period>
  ToDuration duration_cast(const duration<Rep, Period>& d);
```

1 *Requires:* `ToDuration` shall be an instantiation of `duration`. Diagnostic required.

2 *Returns:* Let `CF` be `ratio_divide<Period, typename ToDuration::period>::type`, and `CR` be `common_type< typename ToDuration::rep, Rep, intmax_t>::type`.

— If `CF::num == 1` and `CF::den == 1`, returns

```
ToDuration(static_cast<typename ToDuration::rep>(d.count()))
```

— otherwise, if `CF::num != 1` and `CF::den == 1`, returns

```
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) * static_cast<CR>(CF::num)))
```

— otherwise, if `CF::num == 1` and `CF::den != 1`, returns

```
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) / static_cast<CR>(CF::den)))
```

— otherwise, returns

```
ToDuration(static_cast<typename ToDuration::rep>(
  static_cast<CR>(d.count()) * static_cast<CR>(CF::num) / static_cast<CR>(CF::den)))
```

3 *Remarks:* This function shall not rely on any implicit conversions. All conversions shall be accomplished through `static_cast`. The implementation shall avoid all multiplications or divisions when it is known at compile time that they can be avoided because one or more arguments are 1. All intermediate computations shall be carried out in the widest possible representation and only converted to the destination representation at the final step.

20.8.4 Class template `time_point` [time.point]

```
template <class Clock, class Duration = typename Clock::duration>
class time_point {
public:
  typedef Clock                     clock;
  typedef Duration                  duration;
  typedef typename duration::rep    rep;
  typedef typename duration::period period;
private:
  duration d_;  // exposition only

public:
  // 20.8.4.1, construct
  time_point();  // has value epoch
  explicit time_point(const duration& d);  // same as time_point() + d
  template <class Duration2>
```

```
    time_point(const time_point<clock, Duration2>& t);

  // 20.8.4.2, observer:
  duration time_since_epoch() const;

  // 20.8.4.3, arithmetic:
  time_point& operator+=(const duration& d);
  time_point& operator-=(const duration& d);

  // 20.8.4.4, special values:
  static constexpr time_point min();
  static constexpr time_point max();
};
```

1 `Clock` shall meet the Clock requirements (20.8.5).

2 `Duration` shall be an instance of `duration`. Diagnostic required.

20.8.4.1 `time_point` constructors [time.point.cons]

```
time_point();
```

1 *Effects:* Constructs an object of type `time_point`, initializing `d_` with `duration::zero()`. Such a `time_point` object represents the epoch.

```
time_point(const duration& d);
```

2 *Effects:* Constructs an object of type `time_point`, initializing `d_` with `d`. Such a `time_point` object represents the epoch `+ d`.

```
template <class Duration2>
  time_point(const time_point<clock, Duration2>& t);
```

3 *Requires:* `Duration2` shall be implicitly convertible to `duration`. Diagnostic required.

4 *Effects:* Constructs an object of type `time_point`, initializing `d_` with `t.time_since_epoch()`.

20.8.4.2 `time_point` observer [time.point.observer]

```
duration time_since_epoch() const;
```

1 *Returns:* `d_`.

20.8.4.3 `time_point` arithmetic [time.point.arithmetic]

```
time_point& operator+=(const duration& d);
```

1 *Effects:* `d_ += d`.

2 *Returns:* `*this`.

```
time_point& operator-=(const duration& d);
```

3 *Effects:* `d_ -= d`.

4 *Returns:* `*this`.

20.8.4.4 `time_point` special values [time.point.special]

```
static constexpr time_point min();
```

1 *Returns:* `time_point(duration::min())`.

```
static constexpr time_point max();
```

2 *Returns:* `time_point(duration::max())`.

20.8.4.5 `time_point` non-member arithmetic [time.point.nonmember]

```
template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>{}>::type>
  operator+(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
```

1 *Returns:* `CT(lhs) += rhs`, where `CT` is the type of the return value.

```
template <class Rep1, class Period1, class Clock, class Duration2>
  time_point<Clock, typename common_type<duration<Rep1, Period1>, Duration2>::type>
  operator+(const duration<Rep1, Period1>& lhs, const time_point<Clock, Duration2>& rhs);
```

2 *Returns:* `rhs + lhs`.

```
template <class Clock, class Duration1, class Rep2, class Period2>
  time_point<Clock, typename common_type<Duration1, duration<Rep2, Period2>{}>::type>
  operator-(const time_point<Clock, Duration1>& lhs, const duration<Rep2, Period2>& rhs);
```

3 *Returns:* `lhs + (-rhs)`.

```
template <class Clock, class Duration1, class Duration2>
  typename common_type<Duration1, Duration2>::type
  operator-(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

4 *Returns:* `lhs.time_since_epoch() - rhs.time_since_epoch()`.

20.8.4.6 `time_point` comparisons [time.point.comparisons]

```
template <class Clock, class Duration1, class Duration2>
  bool operator==(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

1 *Returns:* `lhs.time_since_epoch() == rhs.time_since_epoch()`.

```
template <class Clock, class Duration1, class Duration2>
  bool operator!=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

2 *Returns:* `!(lhs == rhs)`.

```
template <class Clock, class Duration1, class Duration2>
  bool operator<(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

3 *Returns:* `lhs.time_since_epoch() < rhs.time_since_epoch()`.

```
template <class Clock, class Duration1, class Duration2>
  bool operator<=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

4 *Returns:* `!(rhs < lhs)`.

```
template <class Clock, class Duration1, class Duration2>
```

```
  bool operator>(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

5 *Returns:* `rhs < lhs`.

```
template <class Clock, class Duration1, class Duration2>
  bool operator>=(const time_point<Clock, Duration1>& lhs, const time_point<Clock, Duration2>& rhs);
```

6 *Returns:* `!(lhs < rhs)`.

20.8.4.7 `time_point_cast` [time.point.cast]

```
template <class ToDuration, class Clock, class Duration>
  time_point<Clock, ToDuration> time_point_cast(const time_point<Clock, Duration>& t);
```

1 *Requires:* `ToDuration` shall be an instance of `duration`. Diagnostic required.

2 *Returns:* `time_point<Clock, ToDuration>(duration_cast<ToDuration>(t.time_since_epoch()))`.

20.8.5 Clocks [time.clock]

1 The types defined in this subclause shall satisfy the Clock requirements (20.8.1).

20.8.5.1 Class `system_clock` [time.clock.system]

1 Objects of class `system_clock` represent wall clock time from the system-wide realtime clock.

```
  class system_clock {
  public:
    typedef see below                       rep;
    typedef ratio<unspecified, unspecified>  period;
    typedef chrono::duration<rep, period>    duration;
    typedef chrono::time_point<system_clock> time_point;
    static const bool is_monotonic =         unspecified ;

    static time_point now();

    // Map to C API
    static time_t      to_time_t  (const time_point& t);
    static time_point  from_time_t(time_t t);
  };
```

2 `system_clock::duration::min() < system_clock::duration::zero()` shall be `true`.

```
time_t to_time_t(const time_point& t);
```

3 *Returns:* A `time_t` object that represents the same point in time as `t` when both values are truncated to the coarser of the precisions of `time_t` and `time_point`.

```
time_point from_time_t(time_t t);
```

4 *Returns:* A `time_point` object that represents the same point in time as `t` when both values are truncated to the coarser of the precisions of `time_t` and `time_point`.

20.8.5.2 Class `monotonic_clock` [time.clock.monotonic]

1 Objects of class `monotonic_clock` represent clocks for which values of `time_point` never decrease as physical time advances. `monotonic_clock` may be a synonym for `system_clock`.

2 The class `monotonic_clock` is conditionally supported.

```
class monotonic_clock {
public:
  typedef unspecified                                  rep;
  typedef ratio<unspecified , unspecified >            period;
  typedef chrono::duration<rep, period>                duration;
  typedef chrono::time_point<unspecified, duration>    time_point;
  static const bool is_monotonic =                     true;

  static time_point now();
};
```

20.8.5.3 Class `high_resolution_clock` [time.clock.hires]

1 Objects of class `high_resolution_clock` represent clocks with the shortest tick period. `high_resolution_clock` may be a synonym for `system_clock` or `monotonic_clock`.

```
class high_resolution_clock {
public:
  typedef unspecified                                   rep;
  typedef ratio<unspecified , unspecified >             period;
  typedef chrono::duration<rep, period>                 duration;
  typedef chrono::time_point<unspecified , duration>    time_point;
  static const bool is_monotonic =                      unspecified ;

  static time_point now();
};
```

20.9 Date and time functions [date.time]

1 Table 46 describes the header `<ctime>`.

Table 46 — Header `<ctime>` synopsis

Type	Name(s)			
Macros:	`NULL`	`CLOCKS_PER_SEC`		
Types:	`size_t`	`clock_t`	`time_t`	
Struct:	`tm`			
Functions:				
`asctime`	`clock`	`difftime`	`localtime`	`strftime`
`ctime`	`gmtime`	`mktime`	`time`	

2 The contents are the same as the Standard C library header `<time.h>`.[227] The functions `asctime`, `ctime`, `gmtime`, and `localtime` are not required to avoid data races (17.6.5.7).

See also: ISO C Clause 7.12, Amendment 1 Clause 4.6.4.

227) `strftime` supports the C99 conversion specifiers `C`, `D`, `e`, `F`, `g`, `G`, `h`, `r`, `R`, `t`, `T`, `u`, `V`, and `z`, and the modifiers `E` and `O`.

21 Strings library [strings]

1 This Clause describes components for manipulating sequences of any literal (3.9) type. In this Clause such types are called *char-like types* , and objects of char-like types are called *char-like objects* or simply *characters*.

2 The following subclauses describe a character traits class, a string class, and null-terminated sequence utilities, as summarized in Table 47.

Table 47 — Strings library summary

Subclause		Header(s)
21.1	Character traits	`<string>`
21.2	String classes	`<string>`
21.5	Null-terminated sequence utilities	`<cctype>` `<cwctype>` `<cstring>` `<cwchar>` `<cstdlib>` `<cuchar>`

21.1 Character traits [char.traits]

1 This subclause defines requirements on classes representing *character traits*, and defines a class template `char_traits<charT>`, along with four specializations, `char_traits<char>`, `char_traits<char16_t>`, `char_traits <char32_t>`, and `char_traits<wchar_t>`, that satisfy those requirements.

2 Most classes specified in Clauses 21.2 and 27 need a set of related types and functions to complete the definition of their semantics. These types and functions are provided as a set of member typedefs and functions in the template parameter 'traits' used by each such template. This subclause defines the semantics guaranteed by these members.

3 To specialize those templates to generate a string or iostream class to handle a particular character container type `CharT`, that and its related character traits class `Traits` are passed as a pair of parameters to the string or iostream template as formal parameters `charT` and `traits`. `Traits::char_type` shall be the same as `CharT`.

4 This subclause specifies a struct template, `char_traits<charT>`, and four explicit specializations of it, `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`, all of which appear in the header `<string>` and satisfy the requirements below.

21.1.1 Character traits requirements [char.traits.require]

1 In Table 48, `X` denotes a Traits class defining types and functions for the character container type `CharT`; `c` and `d` denote values of type `CharT`; `p` and `q` denote values of type `const CharT*`; `s` denotes a value of type `CharT*`; `n`, `i` and `j` denote values of type `std::size_t`; `e` and `f` denote values of type `X::int_type`; `pos` denotes a value of type `X::pos_type`; `state` denotes a value of type `X::state_type`; and `r` denotes an lvalue of type `CharT`. Operations on Traits shall not throw exceptions.

Table 48 — Character traits requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::char_type`	`charT`	(described in 21.1.2)	compile-time
`X::int_type`		(described in 21.1.2)	compile-time
`X::off_type`		(described in 21.1.2)	compile-time
`X::pos_type`		(described in 21.1.2)	compile-time
`X::state_type`		(described in 21.1.2)	compile-time
`X::eq(c,d)`	`bool`	yields: whether `c` is to be treated as equal to `d`.	constant
`X::lt(c,d)`	`bool`	yields: whether `c` is to be treated as less than `d`.	constant
`X::compare(p,q,n)`	`int`	yields: `0` if for each `i` in `[0,n)`, `X::eq(p[i],q[i])` is true; else, a negative value if, for some `j` in `[0,n)`, `X::lt(p[j],q[j])` is true and for each `i` in `[0,j)` `X::eq(p[i],q[i])` is true; else a positive value.	linear
`X::length(p)`	`std::size_t`	yields: the smallest `i` such that `X::eq(p[i],charT())` is true.	linear
`X::find(p,n,c)`	`const X::char_type*`	yields: the smallest `q` in `[p,p+n)` such that `X::eq(*q,c)` is true, zero otherwise.	linear
`X::move(s,p,n)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`. Copies correctly even where the ranges `[p,p+n)` and `[s,s+n)` overlap. yields: `s`.	linear
`X::copy(s,p,n)`	`X::char_type*`	pre: `p` not in `[s,s+n)`. yields: `s`. for each `i` in `[0,n)`, performs `X::assign(s[i],p[i])`.	linear
`X::assign(r,d)`	(not used)	assigns `r=d`.	constant
`X::assign(s,n,c)`	`X::char_type*`	for each `i` in `[0,n)`, performs `X::assign(s[i],c)`. yields: `s`.	linear
`X::not_eof(e)`	`int_type`	yields: `e` if `X::eq_int_type(e,X::eof())` is false, otherwise a value `f` such that `X::eq_int_type(f,X::eof())` is false.	constant
`X::to_char_type(e)`	`X::char_type`	yields: if for some `c`, `X::eq_int_type(e,X::to_int_type(c))` is true, `c`; else some unspecified value.	constant

Table 48 — Character traits requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::to_int_type(c)`	`X::int_type`	yields: some value `e`, constrained by the definitions of `to_char_type` and `eq_int_type`.	constant
`X::eq_int_type(e,f)`	`bool`	yields: for all `c` and `d`, `X::eq(c,d)` is equal to `X::eq_int_type(X::to_int_type(c), X::to_int_type(d))`; otherwise, yields true if `e` and `f` are both copies of `X::eof()`; otherwise, yields false if one of `e` and `f` is a copy of `X::eof()` and the other is not; otherwise the value is unspecified.	constant
`X::eof()`	`X::int_type`	yields: a value `e` such that `X::eq_int_type(e,X::to_int_type(c))` is false for all values `c`.	constant

2 The struct template

```
template<class charT> struct char_traits;
```

shall be provided in the header `<string>` as a basis for explicit specializations.

21.1.2 traits typedefs [char.traits.typedefs]

```
typedef CHAR_T char_type;
```

1 The type `char_type` is used to refer to the character container type in the implementation of the library classes defined in 21.2 and Clause 27.

```
typedef INT_T int_type;
```

2 *Requires:* For a certain character container type `char_type`, a related container type `INT_T` shall be a type or class which can represent all of the valid characters converted from the corresponding `char_type` values, as well as an end-of-file value, `eof()`. The type `int_type` represents a character container type which can hold end-of-file to be used as a return type of the iostream class member functions.[228]

```
typedef OFF_T off_type;
typedef POS_T pos_type;
```

3 *Requires:* Requirements for `off_type` and `pos_type` are described in 27.1.2.

```
typedef STATE_T state_type;
```

4 *Requires:* `state_type` shall meet the requirements of `Assignable` (23.1), `CopyConstructible` (20.1.8), and `DefaultConstructible` types.

228) If `eof()` can be held in `char_type` then some iostreams operations may give surprising results.

21.1.3 `char_traits` specializations [char.traits.specializations]

```
namespace std {
  template<> struct char_traits<char>;
  template<> struct char_traits<char16_t>;
  template<> struct char_traits<char32_t>;
  template<> struct char_traits<wchar_t>;
}
```

1 The header `<string>` shall define four specializations of the template struct `char_traits`: `char_traits<char>`, `char_traits<char16_t>`, `char_traits<char32_t>`, and `char_traits<wchar_t>`.

2 The requirements for the members of these specializations are given in Clause 21.1.1.

21.1.3.1 `struct char_traits<char>` [char.traits.specializations.char]

```
namespace std {
  template<> struct char_traits<char> {
    typedef char        char_type;
    typedef int         int_type;
    typedef streamoff   off_type;
    typedef streampos   pos_type;
    typedef mbstate_t   state_type;

    static void assign(char_type& c1, const char_type& c2);
    static constexpr bool eq(char_type c1, char_type c2);
    static constexpr bool lt(char_type c1, char_type c2);

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c);
    static constexpr char_type to_char_type(int_type c);
    static constexpr int_type to_int_type(char_type c);
    static constexpr bool eq_int_type(int_type c1, int_type c2);
    static constexpr int_type eof();
  };
}
```

1 The defined types for `int_type`, `pos_type`, `off_type`, and `state_type` shall be `int`, `streampos`, `streamoff`, and `mbstate_t` respectively.

2 The type `streampos` shall be an implementation-defined type that satisfies the requirements for `POS_T` in 21.1.2.

3 The type `streamoff` shall be an implementation-defined type that satisfies the requirements for `OFF_T` in 21.1.2.

4 The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

5 The two-argument member `assign` shall be defined identically to the built-in operator `=`. The two-argument members `eq` and `lt` shall be defined identically to the built-in operators `==` and `<` for type `unsigned char`.

6 The member `eof()` shall return `EOF`.

21.1.3.2 `struct char_traits<char16_t>` [char.traits.specializations.char16_t]

```
namespace std {
  template<> struct char_traits<char16_t> {
    typedef char16_t        char_type;
    typedef uint_least16_t  int_type;
    typedef streamoff       off_type;
    typedef u16streampos    pos_type;
    typedef mbstate_t       state_type;

    static void assign(char_type& c1, const char_type& c2);
    static constexpr bool eq(char_type c1, char_type c2);
    static constexpr bool lt(char_type c1, char_type c2);

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c);
    static constexpr char_type to_char_type(int_type c);
    static constexpr int_type to_int_type(char_type c);
    static constexpr bool eq_int_type(int_type c1, int_type c2);
    static constexpr int_type eof();
  };
}
```

1 The type `u16streampos` shall be an implementation-defined type that satisfies the requirements for `POS_T` in 21.1.2.

2 The two-argument members `assign`, `eq`, and `lt` shall be defined identically to the built-in operators `=`, `==`, and `<` respectively.

3 The member `eof()` shall return an implementation-defined constant that cannot appear as a valid UTF-16 code unit.

21.1.3.3 `struct char_traits<char32_t>` [char.traits.specializations.char32_t]

```
namespace std {
  template<> struct char_traits<char32_t> {
    typedef char32_t        char_type;
    typedef uint_least32_t  int_type;
    typedef streamoff       off_type;
    typedef u32streampos    pos_type;
    typedef mbstate_t       state_type;

    static void assign(char_type& c1, const char_type& c2);
    static constexpr bool eq(char_type c1, char_type c2);
    static constexpr bool lt(char_type c1, char_type c2);
```

```
    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c);
    static constexpr char_type to_char_type(int_type c);
    static constexpr int_type to_int_type(char_type c);
    static constexpr bool eq_int_type(int_type c1, int_type c2);
    static constexpr int_type eof();
  };
}
```

1 The type `u32streampos` shall be an implementation-defined type that satisfies the requirements for `POS_T` in 21.1.2.

2 The two-argument members `assign`, `eq`, and `lt` shall be defined identically to the built-in operators `=`, `==`, and `<` respectively.

3 The member `eof()` shall return an implementation-defined constant that cannot appear as a Unicode code point.

21.1.3.4 `struct char_traits<wchar_t>` [char.traits.specializations.wchar.t]

```
namespace std {
  template<> struct char_traits<wchar_t> {
    typedef wchar_t      char_type;
    typedef wint_t       int_type;
    typedef streamoff    off_type;
    typedef wstreampos   pos_type;
    typedef mbstate_t    state_type;

    static void assign(char_type& c1, const char_type& c2);
    static constexpr bool eq(char_type c1, char_type c2);
    static constexpr bool lt(char_type c1, char_type c2);

    static int compare(const char_type* s1, const char_type* s2, size_t n);
    static size_t length(const char_type* s);
    static const char_type* find(const char_type* s, size_t n,
                 const char_type& a);
    static char_type* move(char_type* s1, const char_type* s2, size_t n);
    static char_type* copy(char_type* s1, const char_type* s2, size_t n);
    static char_type* assign(char_type* s, size_t n, char_type a);

    static constexpr int_type not_eof(int_type c);
    static constexpr char_type to_char_type(int_type c);
    static constexpr int_type to_int_type(char_type c);
    static constexpr bool eq_int_type(int_type c1, int_type c2);
    static constexpr int_type eof();
  };
}
```

1 The defined types for `int_type`, `pos_type`, and `state_type` shall be `wint_t`, `wstreampos`, and `mbstate_t` respectively.

2 The type `wstreampos` shall be an implementation-defined type that satisfies the requirements for `POS_T` in 21.1.2.

3 The type `mbstate_t` is defined in `<cwchar>` and can represent any of the conversion states that can occur in an implementation-defined set of supported multibyte character encoding rules.

4 The two-argument members `assign`, `eq`, and `lt` shall be defined identically to the built-in operators `=`, `==`, and `<` respectively.

5 The member `eof()` shall return `WEOF`.

21.2 String classes [string.classes]

1 The header `<string>` defines the `basic_string` class template for manipulating varying-length sequences of char-like objects and four typedefs, `string`, `u16string`, `u32string`, and `wstring`, that name the specializations `basic_string<char>`, `basic_string<char16_t>`, `basic_string<char32_t>`, and `basic_string<wchar_t>`, respectively.

Header `<string>` synopsis

```
namespace std {
  // 21.1, character traits:
  template<class charT> struct char_traits;
  template <> struct char_traits<char>;
  template <> struct char_traits<char16_t>;
  template <> struct char_traits<char32_t>;
  template <> struct char_traits<wchar_t>;

  // 21.3, basic_string:
  template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
      class basic_string;

  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                const basic_string<charT,traits,Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&&
      operator+(basic_string<charT,traits,Allocator>&& lhs,
                const basic_string<charT,traits,Allocator>& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&&
      operator+(const basic_string<charT,traits,Allocator>& lhs,
                basic_string<charT,traits,Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>&&
      operator+(basic_string<charT,traits,Allocator>&& lhs,
                basic_string<charT,traits,Allocator>&& rhs);
  template<class charT, class traits, class Allocator>
    basic_string<charT,traits,Allocator>
      operator+(const charT* lhs,
                const basic_string<charT,traits,Allocator>& rhs);
  template<class charT, class traits, class Allocator>
```

```
  basic_string<charT,traits,Allocator>&&
    operator+(const charT* lhs,
              basic_string<charT,traits,Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(charT lhs, const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(charT lhs, basic_string<charT,traits,Allocator>&& rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const charT* rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const charT* rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs, charT rhs);
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs, charT rhs);

template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);

template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

```
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);

// 21.3.8.8: swap
template<class charT, class traits, class Allocator>
   void swap(basic_string<charT,traits,Allocator>& lhs,
             basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
   void swap(basic_string<charT,traits,Allocator>&& lhs,
             basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
   void swap(basic_string<charT,traits,Allocator>& lhs,
             basic_string<charT,traits,Allocator>&& rhs);

// 21.3.8.9: inserters and extractors
template<class charT, class traits, class Allocator>
 basic_istream<charT,traits>&
  operator>>(basic_istream<charT,traits>&& is,
             basic_string<charT,traits,Allocator>& str);
template<class charT, class traits, class Allocator>
 basic_ostream<charT, traits>&
  operator<<(basic_ostream<charT, traits>&& os,
             const basic_string<charT,traits,Allocator>& str);
template<class charT, class traits, class Allocator>
 basic_istream<charT,traits>&
   getline(basic_istream<charT,traits>&& is,
           basic_string<charT,traits,Allocator>& str,
           charT delim);
template<class charT, class traits, class Allocator>
 basic_istream<charT,traits>&
   getline(basic_istream<charT,traits>&& is,
           basic_string<charT,traits,Allocator>& str);
```

```
  // basic_string typedef names
  typedef basic_string<char> string;
  typedef basic_string<char16_t> u16string;
  typedef basic_string<char32_t> u32string;
  typedef basic_string<wchar_t> wstring;

  // 21.4: numeric conversions
  int stoi(const string& str, size_t *idx = 0, int base = 10);
  long stol(const string& str, size_t *idx = 0, int base = 10);
  unsigned long stoul(const string& str, size_t *idx = 0, int base = 10);
  long long stoll(const string& str, size_t *idx = 0, int base = 10);
  unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10);
  float stof(const string& str, size_t *idx = 0);
  double stod(const string& str, size_t *idx = 0);
  long double stold(const string& str, size_t *idx = 0);
  string to_string(long long val);
  string to_string(unsigned long long val);
  string to_string(long double val);

  int stoi(const wstring& str, size_t *idx = 0, int base = 10);
  long stol(const wstring& str, size_t *idx = 0, int base = 10);
  unsigned long stoul(const wstring& str, size_t *idx = 0, int base = 10);
  long long stoll(const wstring& str, size_t *idx = 0, int base = 10);
  unsigned long long stoull(const wstring& str, size_t *idx = 0, int base = 10);
  float stof(const wstring& str, size_t *idx = 0);
  double stod(const wstring& str, size_t *idx = 0);
  long double stold(const wstring& str, size_t *idx = 0);
  wstring to_wstring(long long val);
  wstring to_wstring(unsigned long long val);
  wstring to_wstring(long double val);
}
```

21.3 Class template `basic_string` [basic.string]

1 The class template `basic_string` describes objects that can store a sequence consisting of a varying number of arbitrary char-like objects with the first element of the sequence at position zero. Such a sequence is also called a "string" if the type of the char-like objects that it holds is clear from context. In the rest of this Clause, the type of the char-like objects held in a `basic_string` object is designated by `charT`.

2 The member functions of `basic_string` use an object of the `Allocator` class passed as a template parameter to allocate and free storage for the contained char-like objects. [229]

3 The class template `basic_string` conforms to the requirements for a Sequence Container (23.1.3), for a Reversible Container (23.1), and for an Allocator-aware container (82). The iterators supported by `basic_string` are random access iterators (24.1.6).

4 In all cases, `size() <= capacity()`.

5 The functions described in this Clause can report two kinds of errors, each associated with an exception type:

— a *length* error is associated with exceptions of type `length_error` (19.1.4);

— an *out-of-range* error is associated with exceptions of type `out_of_range` (19.1.5).

229) [*Note:* `Allocator::value_type` must name the same type as `charT` (21.3.1). — *end note*]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
    class Allocator = allocator<charT> >
  class basic_string {
  public:
    // types:
    typedef          traits                    traits_type;
    typedef typename traits::char_type          value_type;
    typedef          Allocator                 allocator_type;
    typedef typename Allocator::size_type       size_type;
    typedef typename Allocator::difference_type difference_type;

    typedef typename Allocator::reference       reference;
    typedef typename Allocator::const_reference const_reference;
    typedef typename Allocator::pointer         pointer;
    typedef typename Allocator::const_pointer   const_pointer;

    typedef implementation-defined              iterator;       // See 23.1
    typedef implementation-defined              const_iterator; // See 23.1
    typedef std::reverse_iterator<iterator> reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    static const size_type npos = -1;

    // 21.3.2 construct/copy/destroy:
    explicit basic_string(const Allocator& a = Allocator());
    basic_string(const basic_string& str);
    basic_string(basic_string&& str);
    basic_string(const basic_string& str, size_type pos, size_type n = npos,
                 const Allocator& a = Allocator());
    basic_string(const charT* s,
                 size_type n, const Allocator& a = Allocator());
    basic_string(const charT* s, const Allocator& a = Allocator());
    basic_string(size_type n, charT c, const Allocator& a = Allocator());
    template<class InputIterator>
      basic_string(InputIterator begin, InputIterator end,
                   const Allocator& a = Allocator());
    basic_string(initializer_list<charT>, const Allocator& = Allocator());
    basic_string(const basic_string&, const Allocator&);
    basic_string(basic_string&&, const Allocator&);

   ~basic_string();
    basic_string& operator=(const basic_string& str);
    basic_string& operator=(basic_string&& str);
    basic_string& operator=(const charT* s);
    basic_string& operator=(charT c);
    basic_string& operator=(initializer_list<charT>);

    // 21.3.3 iterators:
    iterator       begin();
    const_iterator begin() const;
    iterator       end();
    const_iterator end() const;

    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
```

```
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 21.3.4 capacity:
size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const;
void reserve(size_type res_arg = 0);
void shrink_to_fit();
void clear();
bool empty() const;

// 21.3.5 element access:
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
const_reference at(size_type n) const;
reference       at(size_type n);

const charT& front() const;
charT& front();
const charT& back() const;
charT& back();

// 21.3.6 modifiers:
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& operator+=(initializer_list<charT>);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
basic_string& append(initializer_list<charT>);
void push_back(charT c);

basic_string& assign(const basic_string& str);
basic_string& assign(basic_string&& str);
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
```

```
  basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);

basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
void     insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
  void   insert(const_iterator p, InputIterator first, InputIterator last);
void     insert(const_iterator p, initializer_list<charT>);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator p);
iterator erase(const_iterator first, const_iterator last);

void pop_back();

basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str);
basic_string& replace(size_type pos1, size_type n1,
                      const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2,
                      charT c);

basic_string& replace(iterator i1, iterator i2,
          const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s,
                      size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2,
                      size_type n, charT c);
template<class InputIterator>
  basic_string& replace(iterator i1, iterator i2,
                        InputIterator j1, InputIterator j2);
basic_string& replace(iterator, iterator, initializer_list<charT>);

size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string&& str);

// 21.3.7 string operations:
const charT* c_str() const;         // explicit
const charT* data() const;
allocator_type get_allocator() const;

size_type find (const basic_string& str, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (const charT* s, size_type pos = 0) const;
```

```
    size_type find (charT c, size_type pos = 0) const;
    size_type rfind(const basic_string& str, size_type pos = npos) const;
    size_type rfind(const charT* s, size_type pos, size_type n) const;
    size_type rfind(const charT* s, size_type pos = npos) const;
    size_type rfind(charT c, size_type pos = npos) const;

    size_type find_first_of(const basic_string& str,
                            size_type pos = 0) const;
    size_type find_first_of(const charT* s,
                            size_type pos, size_type n) const;
    size_type find_first_of(const charT* s, size_type pos = 0) const;
    size_type find_first_of(charT c, size_type pos = 0) const;
    size_type find_last_of (const basic_string& str,
                            size_type pos = npos) const;
    size_type find_last_of (const charT* s,
                            size_type pos, size_type n) const;
    size_type find_last_of (const charT* s, size_type pos = npos) const;
    size_type find_last_of (charT c, size_type pos = npos) const;

    size_type find_first_not_of(const basic_string& str,
                size_type pos = 0) const;
    size_type find_first_not_of(const charT* s, size_type pos,
                                size_type n) const;
    size_type find_first_not_of(const charT* s, size_type pos = 0) const;
    size_type find_first_not_of(charT c, size_type pos = 0) const;
    size_type find_last_not_of (const basic_string& str,
                                size_type pos = npos) const;
    size_type find_last_not_of (const charT* s, size_type pos,
                                size_type n) const;
    size_type find_last_not_of (const charT* s,
                                size_type pos = npos) const;
    size_type find_last_not_of (charT c, size_type pos = npos) const;

    basic_string substr(size_type pos = 0, size_type n = npos) const;
    int compare(const basic_string& str) const;
    int compare(size_type pos1, size_type n1,
                const basic_string& str) const;
    int compare(size_type pos1, size_type n1,
                const basic_string& str,
                size_type pos2, size_type n2) const;
    int compare(const charT* s) const;
    int compare(size_type pos1, size_type n1,
                const charT* s) const;
    int compare(size_type pos1, size_type n1,
                const charT* s, size_type n2) const;
  };

  template <class charT, class traits, class Alloc
    struct constructible_with_allocator_suffix<
      basic_string<charT, traits, Alloc> > : true_type { };
}
```

21.3.1 `basic_string` general requirements [string.require]

1 If any operation would cause `size()` to exceed `max_size()`, that operation shall throw an exception object of type `length_error`.

2 In every specialization `basic_string<charT, traits, Allocator>`, the nested type `Allocator::value_type` shall name the same type as `charT`. Every object of type `basic_string<charT, traits, Allocator>` shall use an object of type `Allocator` to allocate and free storage for the contained `charT` objects as needed. The `Allocator` object used shall be a copy of the `Allocator` object passed to the `basic_string` object's constructor or, if the constructor does not take an `Allocator` argument, a copy of a default-constructed `Allocator` object.

3 The char-like objects in a `basic_string` object shall be stored contiguously. That is, for any `basic_string` object `s`, the identity `&*(s.begin() + n) == &*s.begin() + n` shall hold for all values of `n` such that `0 <= n < s.size()`.

4 References, pointers, and iterators referring to the elements of a `basic_string` sequence may be invalidated by the following uses of that `basic_string` object:

— as an argument to any standard library function taking a reference to non-const `basic_string` as an argument.[230]

— Calling non-const member functions, except `operator[]`, `at`, `front`, `back`, `begin`, `rbegin`, `end`, and `rend`.

21.3.2 `basic_string` constructors and assigment operators [string.cons]

```
explicit basic_string(const Allocator& a = Allocator());
```

1 *Effects:* Constructs an object of class `basic_string`. The postconditions of this function are indicated in Table 49.

Table 49 — `basic_string(const Allocator&)` effects

Element	Value
`data()`	a non-null pointer that is copyable and can have 0 added to it
`size()`	0
`capacity()`	an unspecified value

```
basic_string(const basic_string<charT,traits,Allocator>& str);
basic_string(basic_string<charT,traits,Allocator>&& str);
```

2 *Effects:* Constructs an object of class `basic_string` as indicated in Table 50. In the first form, the stored `Allocator` value is copied from `str.get_allocator()`. In the second form, the stored `Allocator` value is move constructed from `str.get_allocator()`, and `str` is left in a valid state with an unspecified value.

3 *Throws:* The second form throws nothing if the allocator's move constructor throws nothing.

```
basic_string(const basic_string<charT,traits,Allocator>& str,
             size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
```

230) For example, as an argument to non-member functions `swap()` (21.3.8.8), `operator>>()` (21.3.8.9), and `getline()` (21.3.8.9), or as an argument to `basic_string::swap()`

Table 50 — `basic_string(const basic_string&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `str.data()`
`size()`	`str.size()`
`capacity()`	a value at least as large as `size()`

4 *Requires:* `pos <= str.size()`

5 *Throws:* `out_of_range` if `pos > str.size()`.

6 *Effects:* Constructs an object of class `basic_string` and determines the effective length `rlen` of the initial string value as the smaller of `n` and `str.size() - pos`, as indicated in Table 51.

Table 51 — `basic_string(const basic_string&, size_type, size_type, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of `rlen` consecutive elements of the string controlled by `str` beginning at position `pos`
`size()`	`rlen`
`capacity()`	a value at least as large as `size()`

```
basic_string(const charT* s, size_type n,
             const Allocator& a = Allocator());
```

7 *Requires:* `s` shall not be a null pointer and `n < npos`.

8 *Effects:* Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `n` whose first element is designated by `s`, as indicated in Table 52.

Table 52 — `basic_string(const charT*, size_type, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `s`
`size()`	`n`
`capacity()`	a value at least as large as `size()`

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

9 *Requires:* `s` shall not be a null pointer.

10 *Effects:* Constructs an object of class `basic_string` and determines its initial string value from the array of `charT` of length `traits::length(s)` whose first element is designated by `s`, as indicated in Table 53.

11 *Remarks:* Uses `traits::length()`.

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

12 *Requires:* `n < npos`

Table 53 — `basic_string(const charT*, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `s`
`size()`	`traits::length(s)`
`capacity()`	a value at least as large as `size()`

13 *Effects:* Constructs an object of class `basic_string` and determines its initial string value by repeating the char-like object `c` for all `n` elements, as indicated in Table 54.

Table 54 — `basic_string(size_t, charT, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated array of `n` elements, each storing the initial value `c`
`size()`	`n`
`capacity()`	a value at least as large as `size()`

```
template<class InputIterator>
  basic_string(InputIterator begin, InputIterator end,
               const Allocator& a = Allocator());
```

14 *Effects:* If `InputIterator` is an integral type, equivalent to

```
basic_string(static_cast<size_type>(begin), static_cast<value_type>(end), a)
```

15 Otherwise constructs a string from the values in the range [`begin`, `end`), as indicated in the Sequence Requirements table (see 23.1.3).

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

16 *Effects:* Same as `basic_string(il.begin(), il.end(), a)`.

```
basic_string(const basic_string& str, const Allocator& alloc);
basic_string(basic_string&& str, const Allocator& alloc);
```

Effects: Constructs an object of class `basic_string` as indicated in Table 55. The stored allocator is constructed from `alloc`. In the second form, `str` is left in a valid state with an unspecified value.

Table 55 — `basic_string(const basic_string&, const Allocator&)` and `basic_string(basic_string&&, const Allocator&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by the original value of `str.data()`.
`size()`	the original value of `str.size()`
`capacity()`	a value at least as large as `size()`
`get_allocator()`	`alloc`

17 *Throws:* The second form throws nothing if `alloc == str.get_allocator()` unless the copy constructor for `Allocator` throws.

```
basic_string<charT,traits,Allocator>&
  operator=(const basic_string<charT,traits,Allocator>& str);
```

18 *Effects:* If `*this` and `str` are not the same object, modifies `*this` as shown in Table 56.

19 If `*this` and `str` are the same object, the member has no effect.

20 *Returns:* `*this`

Table 56 — `operator=(const basic_string<charT, traits, Allocator>&)` effects

Element	Value
`data()`	points at the first element of an allocated copy of the array whose first element is pointed at by `str.data()`
`size()`	`str.size()`
`capacity()`	a value at least as large as `size()`

```
basic_string<charT,traits,Allocator>&
  operator=(const basic_string<charT,traits,Allocator>&& str);
```

21 *Effects:* If `*this` and `str` are not the same object, modifies `*this` as shown in Table 57. The constructor leaves `str` in a valid but unspecified state. [*Note:* A valid implementation is `swap(str)`. — *end note*]

22 If `*this` and `str` are the same object, the member has no effect.

23 *Throws:* Nothing.

24 *Returns:* `*this`

Table 57 — `operator=(const basic_string<charT, traits, Allocator>&&)` effects

Element	Value
`data()`	points at the array whose first element was pointed at by `str.data()`
`size()`	previous value of `str.size()`
`capacity()`	a value at least as large as `size()`

```
basic_string<charT,traits,Allocator>&
  operator=(const charT* s);
```

25 *Returns:* `*this = basic_string<charT,traits,Allocator>(s)`.

26 *Remarks:* Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>& operator=(charT c);
```

27 *Returns:* `*this = basic_string<charT,traits,Allocator>(1,c)`.

```
basic_string& operator=(initializer_list<charT> il);
```

28 *Effects:* `*this = basic_string(il)`.

29 *Returns:* `*this`.

21.3.3 basic_string iterator support [string.iterators]

```
iterator       begin();
const_iterator begin() const;
```

1 *Returns:* an iterator referring to the first character in the string.

```
iterator       end();
const_iterator end() const;
```

2 *Returns:* an iterator which is the past-the-end value.

```
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
```

3 *Returns:* an iterator which is semantically equivalent to `reverse_iterator(end())`.

```
reverse_iterator       rend();
const_reverse_iterator rend() const;
```

4 *Returns:* an iterator which is semantically equivalent to `reverse_iterator(begin())`.

21.3.4 basic_string capacity [string.capacity]

```
size_type size() const;
```

1 *Returns:* a count of the number of char-like objects currently in the string.

```
size_type length() const;
```

2 *Returns:* `size()`.

```
size_type max_size() const;
```

3 *Returns:* The maximum size of the string.

4 *Remark:* See Container requirements table (23.1).

```
void resize(size_type n, charT c);
```

5 *Requires:* `n <= max_size()`

6 *Throws:* `length_error` if `n > max_size()`.

7 *Effects:* Alters the length of the string designated by `*this` as follows:

- If `n <= size()`, the function replaces the string designated by `*this` with a string of length `n` whose elements are a copy of the initial elements of the original string designated by `*this`.
- If `n > size()`, the function replaces the string designated by `*this` with a string of length `n` whose first `size()` elements are a copy of the original string designated by `*this`, and whose remaining elements are all initialized to `c`.

```
void resize(size_type n);
```

8 *Effects:* `resize(n,charT())`.

```
size_type capacity() const;
```

9 *Returns:* the size of the allocated storage in the string.

```
void reserve(size_type res_arg=0);
```

10 The member function `reserve()` is a directive that informs a `basic_string` object of a planned change in size, so that it can manage the storage allocation accordingly.

11 *Effects:* After `reserve()`, `capacity()` is greater or equal to the argument of `reserve`. [*Note:* Calling `reserve()` with a `res_arg` argument less than `capacity()` is in effect a non-binding shrink request. A call with `res_arg <= size()` is in effect a non-binding shrink-to-fit request. — *end note*]

12 *Throws:* `length_error` if `res_arg > max_size()`.[231]

```
void shrink_to_fit();
```

13 *Remarks:* `shrink_to_fit` is a non-binding request to reduce `capacity()` to `size()`. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*]

```
void clear();
```

14 *Effects:* Behaves as if the function calls:

```
erase(begin(), end());
```

```
bool empty() const;
```

15 *Returns:* `size() == 0`.

21.3.5 `basic_string` element access [string.access]

```
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
```

1 *Returns:* If `pos < size()`, returns `*(begin() + pos)`. Otherwise, if `pos == size()`, the `const` version returns `charT()`. Otherwise, the behavior is undefined.

```
const_reference at(size_type pos) const;
reference       at(size_type pos);
```

2 *Requires:* `pos < size()`

3 *Throws:* `out_of_range` if `pos >= size()`.

4 *Returns:* `operator[](pos)`.

```
const charT& front() const;
charT& front();
```

5 *Requires:* `!empty()`

6 *Effects:* Equivalent to `operator[](0)`.

```
const charT& back() const;
charT& back();
```

7 *Requires:* `!empty()`

8 *Effects:* Equivalent to `operator[](size() - 1)`.

231) `reserve()` uses `Allocator::allocate()` which may throw an appropriate exception.

21.3.6 basic_string modifiers [string.modifiers]

21.3.6.1 basic_string::operator+= [string::op+=]

```
basic_string<charT,traits,Allocator>&
  operator+=(const basic_string<charT,traits,Allocator>& str);
```

1 *Returns:* append(str).

```
basic_string<charT,traits,Allocator>& operator+=(const charT* s);
```

2 *Returns:* *this += basic_string<charT,traits,Allocator>(s).

3 *Remarks:* Uses traits::length().

```
basic_string<charT,traits,Allocator>& operator+=(charT c);
```

4 *Returns:* *this += basic_string<charT,traits,Allocator>(1,c).

```
basic_string& operator+=(initializer_list<charT> il);
```

5 *Returns:* The result of append(il).

21.3.6.2 basic_string::append [string::append]

```
basic_string<charT,traits,Allocator>&
  append(const basic_string<charT,traits>& str);
```

1 *Returns:* append(str, 0, npos).

```
basic_string<charT,traits,Allocator>&
  append(const basic_string<charT,traits>& str, size_type pos, size_type n);
```

2 *Requires:* pos <= str.size()

3 *Throws:* out_of_range if pos > str.size().

4 *Effects:* Determines the effective length rlen of the string to append as the smaller of n and str.size() - pos. The function then throws length_error if size() >= npos - rlen.

Otherwise, the function replaces the string controlled by *this with a string of length size() + rlen whose first size() elements are a copy of the original string controlled by *this and whose remaining elements are a copy of the initial elements of the string controlled by str beginning at position pos.

5 *Returns:* *this.

```
basic_string<charT,traits,Allocator>&
  append(const charT* s, size_type n);
```

6 *Returns:* append(basic_string<charT,traits,Allocator>(s,n)).

```
basic_string<charT,traits,Allocator>& append(const charT* s);
```

7 *Returns:* append(basic_string<charT,traits,Allocator>(s)).

8 *Remarks:* Uses traits::length().

```
basic_string<charT,traits,Allocator>&
  append(size_type n, charT c);
```

9 *Returns:* append(basic_string<charT,traits,Allocator>(n,c)).

```
template<class InputIterator>
  basic_string& append(InputIterator first, InputIterator last);
```

10 *Returns:* `append(basic_string<charT,traits,Allocator>(first,last))`.

```
basic_string& append(initializer_list<charT> il);
```

11 *Returns:* `append(basic_string(il))`.

```
void push_back(charT c)
```

12 *Effects:* Equivalent to `append(static_cast<size_type>(1), c)`.

21.3.6.3 `basic_string::assign` [string::assign]

```
basic_string<charT,traits,Allocator>&
  assign(const basic_string<charT,traits>& str);
```

1 *Returns:* `assign(str, 0, npos)`.

```
basic_string<charT,traits,Allocator>&
  assign(const basic_string<charT,traits>&& str);
```

The function replaces the string controlled by `*this` with a string of length `str.size()` whose elements are a copy of the string controlled by `str`. Leaves `str` in a valid but unspecified state. [*Note:* A valid implementation is `swap(str)`. — *end note*]

2 *Throws:* Nothing.

3 *Returns:* `*this`.

```
basic_string<charT,traits,Allocator>&
  assign(const basic_string<charT,traits>& str, size_type pos,
         size_type n);
```

4 *Requires:* `pos <= str.size()`

5 *Throws:* `out_of_range` if `pos > str.size()`.

6 *Effects:* Determines the effective length `rlen` of the string to assign as the smaller of `n` and `str.size() - pos`.

The function then replaces the string controlled by `*this` with a string of length `rlen` whose elements are a copy of the string controlled by `str` beginning at position `pos`.

7 *Returns:* `*this`.

```
basic_string<charT,traits,Allocator>&
  assign(const charT* s, size_type n);
```

8 *Returns:* `assign(basic_string<charT,traits,Allocator>(s,n))`.

```
basic_string<charT,traits,Allocator>& assign(const charT* s);
```

9 *Returns:* `assign(basic_string<charT, traits, Allocator>(s))`.

10 *Remarks:* Uses `traits::length()`.

```
basic_string& assign(initializer_list<charT> il);
```

11 *Returns:* `assign(basic_string(il))`.

```
basic_string<charT,traits,Allocator>&
  assign(size_type n, charT c);
```

12 *Returns:* `assign(basic_string<charT,traits,Allocator>(n,c))`.

```
template<class InputIterator>
  basic_string& assign(InputIterator first, InputIterator last);
```

13 *Returns:* `assign(basic_string<charT,traits,Allocator>(first,last))`.

21.3.6.4 `basic_string::insert` [string::insert]

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos1,
         const basic_string<charT,traits,Allocator>& str);
```

1 *Returns:* `insert(pos1,str,0,npos)`.

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos1,
         const basic_string<charT,traits,Allocator>& str,
         size_type pos2, size_type n);
```

2 *Requires:* `pos1 <= size()` and `pos2 <= str.size()`

3 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > str.size()`.

4 *Effects:* Determines the effective length `rlen` of the string to insert as the smaller of `n` and `str.size() - pos2`. Then throws `length_error` if `size() >= npos - rlen`.

Otherwise, the function replaces the string controlled by `*this` with a string of length `size() + rlen` whose first `pos1` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `rlen` elements are a copy of the elements of the string controlled by `str` beginning at position `pos2`, and whose remaining elements are a copy of the remaining elements of the original string controlled by `*this`.

5 *Returns:* `*this`.

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos, const charT* s, size_type n);
```

6 *Returns:* `insert(pos,basic_string<charT,traits,Allocator>(s,n))`.

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos, const charT* s);
```

7 *Returns:* `insert(pos,basic_string<charT,traits,Allocator>(s))`.

8 *Remarks:* Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>&
  insert(size_type pos, size_type n, charT c);
```

9 *Returns:* `insert(pos,basic_string<charT,traits,Allocator>(n,c))`.

```
iterator insert(const_iterator p, charT c);
```

10 *Requires:* `p` is a valid iterator on `*this`.

11 *Effects:* inserts a copy of `c` before the character referred to by `p`.

12 *Returns:* an iterator which refers to the copy of the inserted character.

```
void insert(const_iterator p, size_type n, charT c);
```

13 *Requires:* `p` is a valid iterator on `*this`.

14 *Effects:* inserts `n` copies of `c` before the character referred to by `p`.

```
template<class InputIterator>
  void insert(const_iterator p, InputIterator first, InputIterator last);
```

15 *Requires:* `p` is a valid iterator on `*this`. `[first,last)` is a valid range.

16 *Effects:* Equivalent to `insert(p - begin(), basic_string(first, last))`.

```
void insert(const_iterator p, initializer_list<charT> il);
```

17 *Effects:* `insert(p, il.begin(), il.end())`.

21.3.6.5 `basic_string::erase` [string::erase]

```
basic_string<charT,traits,Allocator>&
  erase(size_type pos = 0, size_type n = npos);
```

1 *Requires:* `pos <= size()`

2 *Throws:* `out_of_range` if `pos > size()`.

3 *Effects:* Determines the effective length `xlen` of the string to be removed as the smaller of `n` and `size() - pos`.

4 The function then replaces the string controlled by `*this` with a string of length `size() - xlen` whose first `pos` elements are a copy of the initial elements of the original string controlled by `*this`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `pos + xlen`.

5 *Returns:* `*this`.

```
iterator erase(const_iterator p);
```

6 *Effects:* removes the character referred to by `p`.

7 *Returns:* an iterator which points to the element immediately following `p` prior to the element being erased. If no such element exists, `end()` is returned.

```
iterator erase(const_iterator first, const_iterator last);
```

8 *Requires:* `first` and `last` are valid iterators on `*this`, defining a range `[first,last)`.

9 *Effects:* removes the characters in the range `[first,last)`.

10 *Returns:* an iterator which points to the element pointed to by `last` prior to the other elements being erased. If no such element exists, `end()` is returned.

```
void pop_back();
```

11 *Requires:* `!empty()`

12 *Effects:* Equivalent to `erase(size() - 1, 1)`.

21.3.6.6 basic_string::replace [string::replace]

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos1, size_type n1,
          const basic_string<charT,traits,Allocator>& str);
```

1 *Returns:* `replace(pos1, n1, str, 0, npos)`.

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos1, size_type n1,
          const basic_string<charT,traits,Allocator>& str,
          size_type pos2, size_type n2);
```

2 *Requires:* `pos1 <= size() && pos2 <= str.size()`.

3 *Throws:* `out_of_range` if `pos1 > size()` or `pos2 > str.size()`, or `length_error` if the length of the resulting string would exceed `max_size()` (see below).

4 *Effects:* Determines the effective length `xlen` of the string to be removed as the smaller of `n1` and `size() - pos1`. Also determines the effective length `rlen` of the string to be inserted as the smaller of `n2` and `str.size() - pos2`. If `size() - xlen >= max_size() - rlen`, throws `length_error`. Otherwise, the function replaces the string controlled by `*this` with a string of length `size() - xlen + rlen` whose first `pos1` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `rlen` elements are a copy of the initial elements of the string controlled by `str` beginning at position `pos2`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `pos1 + xlen`.

5 *Returns:* `*this`.

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos, size_type n1, const charT* s, size_type n2);
```

6 *Returns:* `replace(pos,n1,basic_string<charT,traits,Allocator>(s,n2))`.

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos, size_type n1, const charT* s);
```

7 *Returns:* `replace(pos,n1,basic_string<charT,traits,Allocator>(s))`.

8 *Remarks:* Uses `traits::length()`.

```
basic_string<charT,traits,Allocator>&
  replace(size_type pos, size_type n1,
          size_type n2, charT c);
```

9 *Returns:* `replace(pos,n1,basic_string<charT,traits,Allocator>(n2,c))`.

```
basic_string& replace(iterator i1, iterator i2, const basic_string& str);
```

10 *Requires:* The iterators `i1` and `i2` are valid iterators on `*this`, defining a range `[i1,i2)`.

11 *Effects:* Replaces the string controlled by `*this` with a string of length `size() - (i2 - i1) + str.size()` whose first `begin() - i1` elements are a copy of the initial elements of the original string controlled by `*this`, whose next `str.size()` elements are a copy of the string controlled by `str`, and whose remaining elements are a copy of the elements of the original string controlled by `*this` beginning at position `i2`.

12 *Returns:* `*this`.

13 *Remarks:* After the call, the length of the string will be changed by: `str.size() - (i2 - i1)`.

```
basic_string&
  replace(iterator i1, iterator i2, const charT* s, size_type n);
```

Returns: `replace(i1,i2,basic_string(s,n))`.

Remarks: Length change: `n - (i2 - i1)`.

```
basic_string& replace(iterator i1, iterator i2, const charT* s);
```

Returns: `replace(i1,i2,basic_string(s))`.

Remarks: Length change: `traits::length(s) - (i2 - i1)`.

Uses `traits::length()`.

```
basic_string& replace(iterator i1, iterator i2, size_type n,
                      charT c);
```

Returns: `replace(i1,i2,basic_string(n,c))`.

Remarks: Length change: `n - (i2 - i1)`.

```
template<class InputIterator>
  basic_string& replace(iterator i1, iterator i2,
                        InputIterator j1, InputIterator j2);
```

Returns: `replace(i1,i2,basic_string(j1,j2))`.

Remarks: Length change: `j2 - j1 - (i2 - i1)`.

```
basic_string& replace(iterator i1, iterator i2,
                      initializer_list<charT> il);
```

Returns: `replace(i1, i2, il.begin(), il.end())`.

21.3.6.7 `basic_string::copy` [string::copy]

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

Requires: `pos <= size()`

Throws: `out_of_range` if `pos > size()`.

Effects: Determines the effective length `rlen` of the string to copy as the smaller of `n` and `size() - pos`. `s` shall designate an array of at least `rlen` elements.

The function then replaces the string designated by `s` with a string of length `rlen` whose elements are a copy of the string controlled by `*this` beginning at position `pos`.

The function does not append a null object to the string designated by `s`.

Returns: `rlen`.

21.3.6.8 `basic_string::swap` [string::swap]

```
void swap(basic_string<charT,traits,Allocator>&& s);
```

Throws: Nothing.

Postcondition: `*this` contains the same sequence of characters that was in `s`, `s` contains the same sequence of characters that was in `*this`.

3 *Complexity:* constant time.

21.3.7 `basic_string` string operations [string.ops]

21.3.7.1 `basic_string` accessors [string.accessors]

```
const charT* c_str() const;
const charT* data() const;
```

1 *Returns:* A pointer to the initial element of an array of length `size() + 1` whose first `size()` elements equal the corresponding elements of the string controlled by `*this` and whose last element is a null character specified by `charT()`.

2 *Requires:* The program shall not alter any of the values stored in the character array.

```
allocator_type get_allocator() const;
```

3 *Returns:* a copy of the `Allocator` object used to construct the string.

21.3.7.2 `basic_string::find` [string::find]

```
size_type find(const basic_string<charT,traits,Allocator>& str,
               size_type pos = 0) const;
```

1 *Effects:* Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos + str.size() <= size()`;

— `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type find(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find(basic_string<charT,traits,Allocator>(s,n),pos)`.

```
size_type find(const charT* s, size_type pos = 0) const;
```

5 *Returns:* `find(basic_string<charT,traits,Allocator>(s),pos)`.

6 *Remarks:* Uses `traits::length()`.

```
size_type find(charT c, size_type pos = 0) const;
```

7 *Returns:* `find(basic_string<charT,traits,Allocator>(1,c),pos)`.

21.3.7.3 `basic_string::rfind` [string::rfind]

```
size_type rfind(const basic_string<charT,traits,Allocator>& str,
                size_type pos = npos) const;
```

1 *Effects:* Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `xpos <= pos` and `xpos + str.size() <= size()`;

— `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type rfind(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `rfind(basic_string<charT,traits,Allocator>(s,n),pos)`.

```
size_type rfind(const charT* s, size_type pos = npos) const;
```

5 *Returns:* `rfind(basic_string<charT,traits,Allocator>(s),pos)`.

6 *Remarks:* Uses `traits::length()`.

```
size_type rfind(charT c, size_type pos = npos) const;
```

7 *Returns:* `rfind(basic_string<charT,traits,Allocator>(1,c),pos)`.

21.3.7.4 `basic_string::find_first_of` [string::find.first.of]

```
size_type
  find_first_of(const basic_string<charT,traits,Allocator>& str,
                size_type pos = 0) const;
```

1 *Effects:* Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos < size()`;

— `traits::eq(at(xpos), str.at(I))` for some element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type
  find_first_of(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find_first_of(basic_string<charT,traits,Allocator>(s,n),pos)`.

```
size_type find_first_of(const charT* s, size_type pos = 0) const;
```

5 *Returns:* `find_first_of(basic_string<charT,traits,Allocator>(s),pos)`.

6 *Remarks:* Uses `traits::length()`.

```
size_type find_first_of(charT c, size_type pos = 0) const;
```

7 *Returns:* `find_first_of(basic_string<charT,traits,Allocator>(1,c),pos)`.

21.3.7.5 `basic_string::find_last_of` [string::find.last.of]

```
size_type
  find_last_of(const basic_string<charT,traits,Allocator>& str,
               size_type pos = npos) const;
```

1 *Effects:* Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `xpos <= pos` and `xpos < size()`;

— `traits::eq(at(xpos), str.at(I))` for some element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type find_last_of(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find_last_of(basic_string<charT,traits,Allocator>(s,n),pos)`.

```
size_type find_last_of(const charT* s, size_type pos = npos) const;
```

5 *Returns:* `find_last_of(basic_string<charT,traits,Allocator>(s),pos)`.

6 *Remarks:* Uses `traits::length()`.

```
size_type find_last_of(charT c, size_type pos = npos) const;
```

7 *Returns:* `find_last_of(basic_string<charT,traits,Allocator>(1,c),pos)`.

21.3.7.6 `basic_string::find_first_not_of` [string::find.first.not.of]

```
size_type
  find_first_not_of(const basic_string<charT,traits,Allocator>& str,
                    size_type pos = 0) const;
```

1 *Effects:* Determines the lowest position `xpos`, if possible, such that both of the following conditions obtain:

— `pos <= xpos` and `xpos < size()`;

— `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type
  find_first_not_of(const charT* s, size_type pos, size_type n) const;
```

4 *Returns:* `find_first_not_of(basic_string<charT,traits,Allocator>(s,n),pos)`.

```
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
```

5 *Returns:* `find_first_not_of(basic_string<charT,traits,Allocator>(s),pos)`.

6 *Remarks:* Uses `traits::length()`.

```
size_type find_first_not_of(charT c, size_type pos = 0) const;
```

7 *Returns:* `find_first_not_of(basic_string<charT,traits,Allocator>(1,c),pos)`.

21.3.7.7 `basic_string::find_last_not_of` [string::find.last.not.of]

```
size_type
  find_last_not_of(const basic_string<charT,traits,Allocator>& str,
                   size_type pos = npos) const;
```

1 *Effects:* Determines the highest position `xpos`, if possible, such that both of the following conditions obtain:

— `xpos <= pos` and `xpos < size()`;

— `traits::eq(at(xpos), str.at(I))` for no element `I` of the string controlled by `str`.

2 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

3 *Remarks:* Uses `traits::eq()`.

```
size_type find_last_not_of(const charT* s, size_type pos,
                           size_type n) const;
```

4 *Returns:* `find_last_not_of(basic_string<charT,traits,Allocator>(s,n),pos)`.

```
size_type find_last_not_of(const charT* s, size_type pos = npos) const;
```

5 *Returns:* `find_last_not_of(basic_string<charT,traits,Allocator>(s),pos)`.

6 *Remarks:* Uses `traits::length()`.

```
size_type find_last_not_of(charT c, size_type pos = npos) const;
```

7 *Returns:* `find_last_not_of(basic_string<charT,traits,Allocator>(1,c),pos)`.

21.3.7.8 `basic_string::substr` [string::substr]

```
basic_string<charT,traits,Allocator>
  substr(size_type pos = 0, size_type n = npos) const;
```

1 *Requires:* `pos <= size()`

2 *Throws:* `out_of_range` if `pos > size()`.

3 *Effects:* Determines the effective length `rlen` of the string to copy as the smaller of `n` and `size() - pos`.

4 *Returns:* `basic_string<charT,traits,Allocator>(data()+pos,rlen)`.

21.3.7.9 `basic_string::compare` [string::compare]

```
int compare(const basic_string<charT,traits,Allocator>& str) const
```

1 *Effects:* Determines the effective length *rlen* of the strings to compare as the smallest of `size()` and `str.size()`. The function then compares the two strings by calling `traits::compare(data(), str.data(), rlen)`.

2 *Returns:* the nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 58.

Table 58 — `compare()` results

Condition	Return Value
`size() < str.size()`	`< 0`
`size() == str.size()`	`0`
`size() > str.size()`	`> 0`

```
int compare(size_type pos1, size_type n1,
            const basic_string<charT,traits,Allocator>& str) const;
```

3 *Returns:*

```
basic_string<charT,traits,Allocator>(*this,pos1,n1).compare(str).
```

```
int compare(size_type pos1, size_type n1,
            const basic_string<charT,traits,Allocator>& str,
            size_type pos2, size_type n2 ) const;
```

4 *Returns:*

```
basic_string<charT,traits,Allocator>(*this,pos1,n1).compare(
  basic_string<charT,traits,Allocator>(str,pos2,n2)).
```

```
int compare(const charT *s) const;
```

5 *Returns:* `this->compare(basic_string<charT,traits,Allocator>(s))`.

```
int compare(size_type pos, size_type n1,
            const charT *s) const;
```

6 *Returns:*

```
basic_string<charT,traits,Allocator>(*this,pos,n1).compare(
  basic_string<charT,traits,Allocator>(s))
```

```
int compare(size_type pos, size_type n1,
            const charT *s, size_type n2) const;
```

7 *Returns:*

```
basic_string<charT,traits,Allocator>(*this,pos,n1).compare(
  basic_string<charT,traits,Allocator>(s,n2))
```

21.3.8 `basic_string` non-member functions [string.nonmembers]

21.3.8.1 `operator+` [string::op+]

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

1 *Returns:* `basic_string<charT,traits,Allocator>(lhs).append(rhs)`

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `lhs.append(rhs)`

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

3 *Returns:* `rhs.insert(0, lhs)`

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

4 *Returns:* `lhs.append(rhs)` [*Note:* Or equivalently `rhs.insert(0, lhs)` — *end note*]

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const charT* lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

5 *Returns:* `basic_string<charT,traits,Allocator>(lhs) + rhs`.

6 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(const charT* lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

7 *Returns:* `rhs.insert(0, lhs)`.

8 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(charT lhs,
              const basic_string<charT,traits,Allocator>& rhs);
```

9 *Returns:* `basic_string<charT,traits,Allocator>(1,lhs) + rhs`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(charT lhs,
              basic_string<charT,traits,Allocator>&& rhs);
```

10 *Returns:* `rhs.insert(0, 1, lhs)`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
    operator+(const basic_string<charT,traits,Allocator>& lhs,
              const charT* rhs);
```

11 *Returns:* `lhs + basic_string<charT,traits,Allocator>(rhs)`.

12 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              const charT* rhs);
```

13 *Returns:* `lhs.append(rhs)`.

14 *Remarks:* Uses `traits::length()`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>
```

```
operator+(const basic_string<charT,traits,Allocator>& lhs,
          charT rhs);
```

Returns: `lhs + basic_string<charT,traits,Allocator>(1,rhs)`.

```
template<class charT, class traits, class Allocator>
  basic_string<charT,traits,Allocator>&&
    operator+(basic_string<charT,traits,Allocator>&& lhs,
              charT rhs);
```

Returns: `lhs.append(1, rhs)`.

21.3.8.2 `operator==` [string::operator==]

```
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

Returns: `lhs.compare(rhs) == 0`.

```
template<class charT, class traits, class Allocator>
  bool operator==(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

Returns: `basic_string<charT,traits,Allocator>(lhs) == rhs`.

```
template<class charT, class traits, class Allocator>
  bool operator==(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

Returns: `lhs == basic_string<charT,traits,Allocator>(rhs)`.

Remarks: Uses `traits::length()`.

21.3.8.3 `operator!=` [string::op!=]

```
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

Returns: `!(lhs == rhs)`.

```
template<class charT, class traits, class Allocator>
  bool operator!=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

Returns: `basic_string<charT,traits,Allocator>(lhs) != rhs`.

```
template<class charT, class traits, class Allocator>
  bool operator!=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

Returns: `lhs != basic_string<charT,traits,Allocator>(rhs)`.

Remarks: Uses `traits::length()`.

21.3.8.4 operator< [string::op<]

```
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

1 *Returns:* `lhs.compare(rhs) < 0`.

```
template<class charT, class traits, class Allocator>
  bool operator< (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `basic_string<charT,traits,Allocator>(lhs) < rhs`.

```
template<class charT, class traits, class Allocator>
  bool operator< (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs < basic_string<charT,traits,Allocator>(rhs)`.

21.3.8.5 operator> [string::op>]

```
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

1 *Returns:* `lhs.compare(rhs) > 0`.

```
template<class charT, class traits, class Allocator>
  bool operator> (const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `basic_string<charT,traits,Allocator>(lhs) > rhs`.

```
template<class charT, class traits, class Allocator>
  bool operator> (const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs > basic_string<charT,traits,Allocator>(rhs)`.

21.3.8.6 operator<= [string::op<=]

```
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

1 *Returns:* `lhs.compare(rhs) <= 0`.

```
template<class charT, class traits, class Allocator>
  bool operator<=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `basic_string<charT,traits,Allocator>(lhs) <= rhs`.

```
template<class charT, class traits, class Allocator>
  bool operator<=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs <= basic_string<charT,traits,Allocator>(rhs)`.

21.3.8.7 operator>= [string::op>=]

```
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

1 *Returns:* `lhs.compare(rhs) >= 0`.

```
template<class charT, class traits, class Allocator>
  bool operator>=(const charT* lhs,
                  const basic_string<charT,traits,Allocator>& rhs);
```

2 *Returns:* `basic_string<charT,traits,Allocator>(lhs) >= rhs`.

```
template<class charT, class traits, class Allocator>
  bool operator>=(const basic_string<charT,traits,Allocator>& lhs,
                  const charT* rhs);
```

3 *Returns:* `lhs >= basic_string<charT,traits,Allocator>(rhs)`.

21.3.8.8 swap [string.special]

```
template<class charT, class traits, class Allocator>
   void swap(basic_string<charT,traits,Allocator>& lhs,
         basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
   void swap(basic_string<charT,traits,Allocator>&& lhs,
         basic_string<charT,traits,Allocator>& rhs);
template<class charT, class traits, class Allocator>
   void swap(basic_string<charT,traits,Allocator>& lhs,
         basic_string<charT,traits,Allocator>&& rhs);
```

1 *Effects:* `lhs.swap(rhs);`

21.3.8.9 Inserters and extractors [string.io]

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    operator>>(basic_istream<charT,traits>&& is,
               basic_string<charT,traits,Allocator>& str);
```

1 *Effects:* Behaves as a formatted input function (27.6.1.2.1). After constructing a `sentry` object, if the sentry converts to true, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1,c)`. If `is.width()` is greater than zero, the maximum number `n` of characters appended is `is.width()`; otherwise `n` is `str.max_size()`. Characters are extracted and appended until any of the following occurs:

- *n* characters are stored;
- end-of-file occurs on the input sequence;
- `isspace(c,is.getloc())` is true for the next available input character *c*.

2 After the last character (if any) is extracted, `is.width(0)` is called and the `sentry` object `k` is destroyed.

3 If the function extracts no characters, it calls `is.setstate(ios::failbit)`, which may throw `ios_base::failure` (27.4.4.3).

4 *Returns:* `is`

```
template<class charT, class traits, class Allocator>
  basic_ostream<charT, traits>&
    operator<<(basic_ostream<charT, traits>&& os,
               const basic_string<charT,traits,Allocator>& str);
```

5 *Effects:* Behaves as a formatted output function (27.6.2.6.1). After constructing a `sentry` object, if this object returns `true` when converted to a value of type `bool`, determines padding as described in 22.2.2.2.2, then inserts the resulting sequence of characters `seq` as if by calling `os.rdbuf()->sputn(seq, n)`, where `n` is the larger of `os.width()` and `str.size()`; then calls `os.width(0)`.

6 *Returns:* `os`

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str,
            charT delim);
```

7 *Effects:* Behaves as an unformatted input function (27.6.1.3), except that it does not affect the value returned by subsequent calls to `basic_istream<>::gcount()`. After constructing a `sentry` object, if the sentry converts to true, calls `str.erase()` and then extracts characters from `is` and appends them to `str` as if by calling `str.append(1, c)` until any of the following occurs:

— end-of-file occurs on the input sequence (in which case, the `getline` function calls `is.setstate(ios_base::eofbit)`).

— `traits::eq(c, delim)` for the next available input character c (in which case, c is extracted but not appended) (27.4.4.3)

— `str.max_size()` characters are stored (in which case, the function calls `is.setstate(ios_base::failbit))` (27.4.4.3)

8 The conditions are tested in the order shown. In any case, after the last character is extracted, the `sentry` object `k` is destroyed.

9 If the function extracts no characters, it calls `is.setstate(ios_base::failbit)` which may throw `ios_base::failure` (27.4.4.3).

10 *Returns:* `is`.

```
template<class charT, class traits, class Allocator>
  basic_istream<charT,traits>&
    getline(basic_istream<charT,traits>&& is,
            basic_string<charT,traits,Allocator>& str)
```

11 *Returns:* `getline(is,str,is.widen('\n'))`

21.4 Numeric Conversions [string.conversions]

```
int stoi(const string& str, size_t *idx = 0, int base = 10);
long stol(const string& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const string& str, size_t *idx = 0, int base = 10);
long long stoll(const string& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t *idx = 0, int base = 10);
```

1 *Effects:* the first two functions call `strtol(str.c_str(), ptr, base)`, and the last three functions call `strtoul(str.c_str(), ptr, base)`, `strtoll(str.c_str(), ptr, base)`, and `strtoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

2 *Returns:* the converted result.

3 *Throws:* `invalid_argument` if `strtol`, `strtoul`, `strtoll`, or `strtoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

```
float stof(const string& str, size_t *idx = 0);
double stod(const string& str, size_t *idx = 0);
long double stold(const string& str, size_t *idx = 0);
```

4 *Effects:* the first two functions call `strtod(str.c_str(), ptr)` and the third function calls `strtold(str.c_str(), ptr)`. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

5 *Returns:* the converted result.

6 *Throws:* `invalid_argument` if `strtod` or `strtold` reports that no conversion could be performed. Throws `out_of_range` if `strtod` or `strtold` sets `errno` to `ERANGE`.

```
string to_string(long long val);
string to_string(unsigned long long val);
string to_string(long double val);
```

7 *Returns:* each function returns a `string` object holding the character representation of the value of its argument that would be generated by calling `sprintf(buf, fmt, val)` with a format specifier of `"%lld"`, `"%llu"`, or `"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size.

```
int stoi(const wstring& str, size_t *idx = 0, int base = 10);
long stol(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long stoul(const wstring& str, size_t *idx = 0, int base = 10);
long long stoll(const wstring& str, size_t *idx = 0, int base = 10);
unsigned long long stoull(const wstring& str, size_t *idx = 0, int base = 10);
```

8 *Effects:* the first two functions call `wcstol(str.c_str(), ptr, base)`, and the last three functions call `wcstoul(str.c_str(), ptr, base)`, `wcstoll(str.c_str(), ptr, base)`, and `wcstoull(str.c_str(), ptr, base)`, respectively. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

9 *Returns:* the converted result.

10 *Throws:* `invalid_argument` if `wcstol`, `wcstoul`, `wcstoll`, or `wcstoull` reports that no conversion could be performed. Throws `out_of_range` if the converted value is outside the range of representable values for the return type.

```
float stof(const wstring& str, size_t *idx = 0);
```

```
double stod(const wstring& str, size_t *idx = 0);
long double stold(const wstring& str, size_t *idx = 0);
```

11 *Effects:* the first two functions call `wcstod(str.c_str(), ptr)` and the third function calls `wcstold(str.c_str(), ptr)`. Each function returns the converted result, if any. The argument `ptr` designates a pointer to an object internal to the function that is used to determine what to store at `*idx`. If the function does not throw an exception and `idx != 0`, the function stores in `*idx` the index of the first unconverted element of `str`.

12 *Returns:* the converted result.

13 *Throws:* `invalid_argument` if `wcstod` or `wcstold` reports that no conversion could be performed. Throws `out_of_range` if `wcstod` or `wcstold` sets `errno` to `ERANGE`.

```
wstring to_wstring(long long val);
wstring to_wstring(unsigned long long val);
wstring to_wstring(long double val);
```

14 *Returns:* Each function returns a `wstring` object holding the character representation of the value of its argument that would be generated by calling `swprintf(buf, buffsz, fmt, val)` with a format specifier of `L"%lld"`, `L"%llu"`, or `L"%Lf"`, respectively, where `buf` designates an internal character buffer of sufficient size `buffsz`.

21.5 Null-terminated sequence utilities [c.strings]

1 Tables 59, 60, 61, 62,63. and 64 describe headers `<cctype>`, `<cwctype>`, `<cstring>`, `<cwchar>`, `<cstdlib>` (character conversions), and `<cuchar>`, respectively.

2 The contents of these headers shall be the same as the Standard C Library headers `<ctype.h>`, `<wctype.h>`, `<string.h>`, `<wchar.h>`, and `<stdlib.h>` and the C Unicode TR header `<uchar.h>`, respectively, with the following modifications:

3 The headers shall not define the types `char16_t`, `char32_t`, and `wchar_t` (2.11).

4 The function signature `strchr(const char*, int)` shall be replaced by the two declarations:

```
const char* strchr(const char* s, int c);
      char* strchr(      char* s, int c);
```

both of which shall have the same behavior as the original declaration.

5 The function signature `strpbrk(const char*, const char*)` shall be replaced by the two declarations:

```
const char* strpbrk(const char* s1, const char* s2);
      char* strpbrk(      char* s1, const char* s2);
```

both of which shall have the same behavior as the original declaration.

6 The function signature `strrchr(const char*, int)` shall be replaced by the two declarations:

```
const char* strrchr(const char* s, int c);
      char* strrchr(      char* s, int c);
```

both of which shall have the same behavior as the original declaration.

7 The function signature `strstr(const char*, const char*)` shall be replaced by the two declarations:

```
const char* strstr(const char* s1, const char* s2);
      char* strstr(      char* s1, const char* s2);
```

both of which shall have the same behavior as the original declaration.

8 The function signature `memchr(const void*, int, size_t)` shall be replaced by the two declarations:

```
const void* memchr(const void* s, int c, size_t n);
      void* memchr(      void* s, int c, size_t n);
```

both of which shall have the same behavior as the original declaration.

9 The function signature `wcschr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:

```
const wchar_t* wcschr(const wchar_t* s, wchar_t c);
      wchar_t* wcschr(      wchar_t* s, wchar_t c);
```

both of which shall have the same behavior as the original declaration.

10 The function signature `wcspbrk(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:

```
const wchar_t* wcspbrk(const wchar_t* s1, const wchar_t* s2);
      wchar_t* wcspbrk(      wchar_t* s1, const wchar_t* s2);
```

both of which shall have the same behavior as the original declaration.

11 The function signature `wcsrchr(const wchar_t*, wchar_t)` shall be replaced by the two declarations:

```
const wchar_t* wcsrchr(const wchar_t* s, wchar_t c);
      wchar_t* wcsrchr(      wchar_t* s, wchar_t c);
```

both of which shall have the same behavior as the original declaration.

12 The function signature `wcsstr(const wchar_t*, const wchar_t*)` shall be replaced by the two declarations:

```
const wchar_t* wcsstr(const wchar_t* s1, const wchar_t* s2);
      wchar_t* wcsstr(      wchar_t* s1, const wchar_t* s2);
```

both of which shall have the same behavior as the original declaration.

13 The function signature `wmemchr(const wwchar_t*, int, size_t)` shall be replaced by the two declarations:

```
const wchar_t* wmemchr(const wchar_t* s, wchar_t c, size_t n);
      wchar_t* wmemchr(      wchar_t* s, wchar_t c, size_t n);
```

both of which shall have the same behavior as the original declaration.

14 The functions `strerror` and `strtok` are not required to avoid data races (17.6.5.7).

See also: ISO C 7.3, 7.10.7, 7.10.8, and 7.11. Amendment 1 4.4, 4.5, and 4.6.

Table 59 — Header `<cctype>` synopsis

Type	Name(s)			
Functions:				
`isalnum`	`isblank`	`isdigit`	`isprint`	`isupper`
`tolower`	`isalpha`	`isgraph`	`ispunct`	`isxdigit`
`toupper`	`iscntrl`	`islower`	`isspace`	

Table 60 — Header <cwctype> synopsis

Type	Name(s)				
Macro:	WEOF <cwctype>				
Types:	wctrans_t	wctype_t	wint_t <cwctype>		
Functions:					
iswalnum	iswcntrl	iswgraph	iswpunct	iswxdigit	towupper
iswalpha	iswctype	iswlower	iswspace	towctrans	wctrans
iswblank	iswdigit	iswprint	iswupper	towlower	wctype

Table 61 — Header <cstring> synopsis

Type	Name(s)			
Macro:	NULL <cstring>			
Type:	size_t <cstring>			
Functions:				
memchr	strcat	strcspn	strncpy	strtok
memcmp	strchr	strerror	strpbrk	strxfrm
memcpy	strcmp	strlen	strrchr	
memmove	strcoll	strncat	strspn	
memset	strcpy	strncmp	strstr	

Table 62 — Header <cwchar> synopsis

Type	Name(s)				
Macros:	NULL <cwchar>	WCHAR_MAX	WCHAR_MIN	WEOF <cwchar>	
Types:	mbstate_t	wint_t <cwchar>	size_t	tm	
Functions:					
btowc	mbrlen	vfwscanf	wcscpy	wcsspn	wcsxfrm
fgetwc	mbrtowc	vswscanf	wcscspn	wcsstr	wctob
fgetws	mbsinit	vswprintf	wcsftime	wcstod	wmemchr
fputwc	mbsrtowcs	vwprintf	wcslen	wcstof	wmemcmp
fputws	putwc	vwscanf	wcsncat	wcstok	wmemcpy
fwide	putwchar	wcrtomb	wcsncmp	wcstol	wmemmove
fwprintf	swprintf	wcscat	wcsncpy	wcstold	wmemset
fwscanf	swscanf	wcschr	wcspbrk	wcstoll	wprintf
getwc	ungetwc	wcscmp	wcsrchr	wcstoul	wscanf
getwchar	vfwprintf	wcscoll	wcsrtombs	wcstoull	

Table 63 — Header <cstdlib> synopsis

Type	Name(s)		
Macros:	MB_CUR_MAX		
Functions:			
atof	mblen	strtof	strtoul
atoi	mbtowc	strtol	strtoull
atol	mbstowcs	strtold	wctomb
atoll	strtod	strtoll	wcstombs

Table 64 — Header <cuchar> synopsis

Type	Name(s)	
Macros:	__STDC_UTF_16__	__STDC_UTF_32__
Functions:	mbrtoc16	c16rtomb
	mbrtoc32	c32rtomb

22 Localization library [localization]

1 This Clause describes components that C++ programs may use to encapsulate (and therefore be more portable when confronting) cultural differences. The locale facility includes internationalization support for character classification and string collation, numeric, monetary, and date/time formatting and parsing, and message retrieval.

2 The following subclauses describe components for locales themselves, the standard facets, and facilities from the ISO C library, as summarized in Table 65.

Table 65 — Localization library summary

Subclause		Header(s)
22.1	Locales	`<locale>`
22.2	Standard `locale` Categories	
22.3	Standard code conversion facets	`<codecvt>`
22.4	C library locales	`<clocale>`

22.1 Locales [locales]

1 **Header `<locale>` synopsis**

```
namespace std {
  // 22.1.1, locale:
  class locale;
  template <class Facet> const Facet& use_facet(const locale&);
  template <class Facet> bool         has_facet(const locale&) throw();

  // 22.1.3, convenience interfaces:
  template <class charT> bool isspace (charT c, const locale& loc);
  template <class charT> bool isprint (charT c, const locale& loc);
  template <class charT> bool iscntrl (charT c, const locale& loc);
  template <class charT> bool isupper (charT c, const locale& loc);
  template <class charT> bool islower (charT c, const locale& loc);
  template <class charT> bool isalpha (charT c, const locale& loc);
  template <class charT> bool isdigit (charT c, const locale& loc);
  template <class charT> bool ispunct (charT c, const locale& loc);
  template <class charT> bool isxdigit(charT c, const locale& loc);
  template <class charT> bool isalnum (charT c, const locale& loc);
  template <class charT> bool isgraph (charT c, const locale& loc);
  template <class charT> charT toupper(charT c, const locale& loc);
  template <class charT> charT tolower(charT c, const locale& loc);
  template <class Codecvt, class Elem = wchar_t> class wstring_convert;
  template <class Codecvt, class Elem = wchar_t,
     class Tr = char_traits<Elem>> class wbuffer_convert;

  // 22.2.1 and 22.2.1.3, ctype:
  class ctype_base;
  template <class charT> class ctype;
```

```
  template <>            class ctype<char>;              // specialization
  template <class charT> class ctype_byname;
  template <>            class ctype_byname<char>;       // specialization
  class codecvt_base;
  template <class internT, class externT, class stateT> class codecvt;
  template <class internT, class externT, class stateT> class codecvt_byname;

  // 22.2.2 and 22.2.3, numeric:
  template <class charT, class InputIterator>  class num_get;
  template <class charT, class OutputIterator> class num_put;
  template <class charT> class numpunct;
  template <class charT> class numpunct_byname;

  // 22.2.4, collation:
  template <class charT> class collate;
  template <class charT> class collate_byname;

  // 22.2.5, date and time:
  class time_base;
  template <class charT, class InputIterator>  class time_get;
  template <class charT, class InputIterator>  class time_get_byname;
  template <class charT, class OutputIterator> class time_put;
  template <class charT, class OutputIterator> class time_put_byname;

  // 22.2.6, money:
  class money_base;
  template <class charT, class InputIterator>  class money_get;
  template <class charT, class OutputIterator> class money_put;
  template <class charT, bool Intl> class moneypunct;
  template <class charT, bool Intl> class moneypunct_byname;

  // 22.2.7, message retrieval:
  class messages_base;
  template <class charT> class messages;
  template <class charT> class messages_byname;
}
```

2 The header <locale> defines classes and declares functions that encapsulate and manipulate the information peculiar to a locale.[232]

22.1.1 Class locale [locale]

```
namespace std {
  class locale {
  public:
    // types:
    class facet;
    class id;
    typedef int category;
    static const category    // values assigned here are for exposition only
      none     = 0,
      collate  = 0x010, ctype    = 0x020,
      monetary = 0x040, numeric  = 0x080,
      time     = 0x100, messages = 0x200,
```

232) In this subclause, the type name struct tm is an incomplete type that is defined in <ctime>.

```
      all = collate | ctype | monetary | numeric | time  | messages;

    // construct/copy/destroy:
    locale() throw();
    locale(const locale& other) throw();
    explicit locale(const char* std_name);
    explicit locale(const string& std_name);
    locale(const locale& other, const char* std_name, category);
    locale(const locale& other, const string& std_name, category);
    template <class Facet> locale(const locale& other, Facet* f);
    locale(const locale& other, const locale& one, category);
   ~locale() throw();           // non-virtual
    const locale& operator=(const locale& other) throw();
    template <class Facet> locale combine(const locale& other) const;

    // locale operations:
    basic_string<char>                 name() const;

    bool operator==(const locale& other) const;
    bool operator!=(const locale& other) const;

    template <class charT, class Traits, class Allocator>
      bool operator()(const basic_string<charT,Traits,Allocator>& s1,
                      const basic_string<charT,Traits,Allocator>& s2) const;

    // global locale objects:
    static       locale  global(const locale&);
    static const locale& classic();
  };
}
```

1 Class `locale` implements a type-safe polymorphic set of facets, indexed by facet *type*. In other words, a facet has a dual role: in one sense, it's just a class interface; at the same time, it's an index into a locale's set of facets.

2 Access to the facets of a `locale` is via two function templates, `use_facet<>` and `has_facet<>`.

3 [*Example:* An iostream `operator<<` might be implemented as:[233]

```
template <class charT, class traits>
basic_ostream<charT,traits>&
operator<< (basic_ostream<charT,traits>& s, Date d) {
  typename basic_ostream<charT,traits>::sentry cerberos(s);
  if (cerberos) {
    ios_base::iostate err = 0;
    tm tmbuf; d.extract(tmbuf);
    use_facet< time_put<charT,ostreambuf_iterator<charT,traits> > >(
      s.getloc()).put(s, s, s.fill(), err, &tmbuf, 'x');
    s.setstate(err);            // might throw
  }
  return s;
}
```

— *end example*]

233) Notice that, in the call to `put`, the stream is implicitly converted to an `ostreambuf_iterator<charT,traits>`.

4 In the call to `use_facet<Facet>(loc)`, the type argument chooses a facet, making available all members of the named type. If `Facet` is not present in a locale, it throws the standard exception `bad_cast`. A C++ program can check if a locale implements a particular facet with the function template `has_facet<Facet>()`. User-defined facets may be installed in a locale, and used identically as may standard facets (22.2.8).

5 [*Note:* All locale semantics are accessed via `use_facet<>` and `has_facet<>`, except that:

— A member operator template `operator()(const basic_string<C,T,A>&, const basic_string<C,T,A>&)` is provided so that a locale may be used as a predicate argument to the standard collections, to collate strings.

— Convenient global interfaces are provided for traditional `ctype` functions such as `isdigit()` and `isspace()`, so that given a locale object `loc` a C++ program can call `isspace(c,loc)`. (This eases upgrading existing extractors (27.6.1.2).) — *end note*]

6 Once a facet reference is obtained from a locale object by calling `use_facet<>`, that reference remains usable, and the results from member functions of it may be cached and re-used, as long as some locale object refers to that facet.

7 In successive calls to a locale facet member function on a facet object installed in the same locale, the returned result shall be identical.

8 A `locale` constructed from a name string (such as `"POSIX"`), or from parts of two named locales, has a name; all others do not. Named locales may be compared for equality; an unnamed locale is equal only to (copies of) itself. For an unnamed locale, `locale::name()` returns the string `"*"`.

9 Whether there is one global locale object for the entire program or one global locale object per thread is implementation defined. Implementations are encouraged but not required to provide one global locale object per thread. If there is a single global locale object for the entire program, implementations are not required to avoid data races on it (17.6.5.7).

22.1.1.1 `locale` types [locale.types]

22.1.1.1.1 Type `locale::category` [locale.category]

```
typedef int category;
```

1 *Valid* `category` values include the `locale` member bitmask elements `collate`, `ctype`, `monetary`, `numeric`, `time`, and `messages`, each of which represents a single locale category. In addition, `locale` member bitmask constant `none` is defined as zero and represents no category. And `locale` member bitmask constant `all` is defined such that the expression

```
(collate | ctype | monetary | numeric | time | messages | all) == all
```

is `true`, and represents the union of all categories. Further, the expression `(X | Y)`, where `X` and `Y` each represent a single category, represents the union of the two categories.

2 `locale` member functions expecting a `category` argument require one of the `category` values defined above, or the union of two or more such values. Such a `category` value identifies a set of locale categories. Each locale category, in turn, identifies a set of locale facets, including at least those shown in Table 66.

3 For any locale `loc` either constructed, or returned by `locale::classic()`, and any facet `Facet` shown in Table 66, `has_facet<Facet>(loc)` is true. Each `locale` member function which takes a `locale::category` argument operates on the corresponding set of facets.

Table 66 — Locale category facets

Category	Includes facets
collate	`collate<char>`, `collate<wchar_t>`

ctype	`ctype<char>`, `ctype<wchar_t>`
	`codecvt<char,char,mbstate_t>`
	`codecvt<char16_t,char,mbstate_t>`
	`codecvt<char32_t,char,mbstate_t>`
	`codecvt<wchar_t,char,mbstate_t>`

monetary	`moneypunct<char>`, `moneypunct<wchar_t>`
	`moneypunct<char,true>`, `moneypunct<wchar_t,true>`
	`money_get<char>`, `money_get<wchar_t>`
	`money_put<char>`, `money_put<wchar_t>`

numeric	`numpunct<char>`, `numpunct<wchar_t>`
	`num_get<char>`, `num_get<wchar_t>`
	`num_put<char>`, `num_put<wchar_t>`

time	`time_get<char>`, `time_get<wchar_t>` `time_put<char>`, `time_put<wchar_t>`
messages	`messages<char>`, `messages<wchar_t>`

Table 67 — Required specializations

Category	Includes facets
collate	`collate_byname<char>`, `collate_byname<wchar_t>`

ctype	`ctype_byname<char>`, `ctype_byname<wchar_t>`
	`codecvt_byname<char,char,mbstate_t>`
	`codecvt_byname<char16_t,char,mbstate_t>`
	`codecvt_byname<char32_t,char,mbstate_t>`
	`codecvt_byname<wchar_t,char,mbstate_t>`

monetary	`moneypunct_byname<char,International>`
	`moneypunct_byname<wchar_t,International>`
	`money_get<C,InputIterator>`
	`money_put<C,OutputIterator>`

numeric	`numpunct_byname<char>`, `numpunct_byname<wchar_t>` `num_get<C,InputIterator>`, `num_put<C,OutputIterator>`

time	`time_get<char,InputIterator>`
	`time_get_byname<char,InputIterator>`
	`time_get<wchar_t,InputIterator>`
	`time_get_byname<wchar_t,InputIterator>`
	`time_put<char,OutputIterator>`
	`time_put_byname<char,OutputIterator>`
	`time_put<wchar_t,OutputIterator>`
	`time_put_byname<wchar_t,OutputIterator>`

messages	`messages_byname<char>`, `messages_byname<wchar_t>`

4 An implementation is required to provide those specializations for facet templates identified as members of a category, and for those shown in Table 67.

5 The provided implementation of members of facets `num_get<charT>` and `num_put<charT>` calls `use_facet<F>(l)` only for facet `F` of types `numpunct<charT>` and `ctype<charT>`, and for locale `l` the value obtained by calling member `getloc()` on the `ios_base&` argument to these functions.

6 In declarations of facets, a template formal parameter with name `InputIterator` or `OutputIterator` indicates the set of all possible specializations on parameters that satisfy the requirements of an Input Iterator or an Output Iterator, respectively (24.1). A template formal parameter with name `C` represents the set of all possible specializations on a parameter that satisfies the requirements for a character on which any of the iostream components can be instantiated. A template formal parameter with name `International` represents the set of all possible specializations on a bool parameter.

22.1.1.1.2 Class `locale::facet` [locale.facet]

```
namespace std {
  class locale::facet {
  protected:
    explicit facet(size_t refs = 0);
    virtual ~facet();
    facet(const facet&) = delete;
    void operator=(const facet&) = delete;
  };
}
```

1 Template parameters in this Clause which are required to be facets are those named `Facet` in declarations. A program that passes a type that is *not* a facet, or a type that refers to a volatile-qualified facet, as an (explicit or deduced) template parameter to a locale function expecting a facet, is ill-formed. A const-qualified facet is a valid template argument to any locale function that expects a Facet template parameter.

2 The `refs` argument to the constructor is used for lifetime management.

— For `refs == 0`, the implementation performs `delete static_cast<locale::facet*>(f)` (where `f` is a pointer to the facet) when the last `locale` object containing the facet is destroyed; for `refs == 1`, the implementation never destroys the facet.

3 Constructors of all facets defined in this Clause take such an argument and pass it along to their `facet` base class constructor. All one-argument constructors defined in this Clause are *explicit*, preventing their participation in automatic conversions.

4 For some standard facets a standard "`..._byname`" class, derived from it, implements the virtual function semantics equivalent to that facet of the locale constructed by `locale(const char*)` with the same name. Each such facet provides a constructor that takes a `const char*` argument, which names the locale, and a `refs` argument, which is passed to the base class constructor. Each such facet also provides a constructor that takes a `string` argument `str` and a `refs` argument, which has the same effect as calling the first constructor with the two arguments `str.c_str()` and `refs`. If there is no "`..._byname`" version of a facet, the base class implements named locale semantics itself by reference to other facets.

22.1.1.1.3 Class `locale::id` [locale.id]

```
namespace std {
  class locale::id {
  public:
    id();
    void operator=(const id&) = delete;
```

```
    id(const id&) = delete;
  };
}
```

1 The class locale::id provides identification of a locale facet interface, used as an index for lookup and to encapsulate initialization.

2 [*Note:* Because facets are used by iostreams, potentially while static constructors are running, their initialization cannot depend on programmed static initialization. One initialization strategy is for `locale` to initialize each facet's `id` member the first time an instance of the facet is installed into a locale. This depends only on static storage being zero before constructors run (3.6.2). — *end note*]

22.1.1.2 `locale` constructors and destructor [locale.cons]

```
locale() throw();
```

1 Default constructor: a snapshot of the current global locale.

2 *Effects:* Constructs a copy of the argument last passed to `locale::global(locale&)`, if it has been called; else, the resulting facets have virtual function semantics identical to those of `locale::classic()`. [*Note:* This constructor is commonly used as the default value for arguments of functions that take a `const locale&` argument. — *end note*]

```
locale(const locale& other) throw();
```

3 *Effects:* Constructs a locale which is a copy of `other`.

```
const locale& operator=(const locale& other) throw();
```

4 *Effects:* Creates a copy of `other`, replacing the current value.

5 *Returns:* `*this`

```
explicit locale(const char* std_name);
```

6 *Effects:* Constructs a locale using standard C locale names, e.g. `"POSIX"`. The resulting locale implements semantics defined to be associated with that name.

7 *Throws:* `runtime_error` if the argument is not valid, or is null.

8 *Remarks:* The set of valid string argument values is `"C"`, `""`, and any implementation-defined values.

```
explicit locale(const string& std_name);
```

9 *Effects:* The same as `locale(std_name.c_str())`.

```
locale(const locale& other, const char* std_name, category);
```

10 *Effects:* Constructs a locale as a copy of `other` except for the facets identified by the `category` argument, which instead implement the same semantics as `locale(std_name)`.

11 *Throws:* `runtime_error` if the argument is not valid, or is null.

12 *Remarks:* The locale has a name if and only if `other` has a name.

```
locale(const locale& other, const string& std_name, category cat);
```

13 *Effects:* The same as `locale(other, std_name.c_str(), cat)`.

```
template <class Facet> locale(const locale& other, Facet* f);
```

14 *Effects:* Constructs a locale incorporating all facets from the first argument except that of type `Facet`, and installs the second argument as the remaining facet. If `f` is null, the resulting object is a copy of `other`.

15 *Remarks:* The resulting locale has no name.

```
locale(const locale& other, const locale& one, category cats);
```

16 *Effects:* Constructs a locale incorporating all facets from the first argument except those that implement `cats`, which are instead incorporated from the second argument.

17 *Remarks:* The resulting locale has a name if and only if the first two arguments have names.

```
~locale() throw();
```

18 A non-virtual destructor that throws no exceptions.

22.1.1.3 `locale` members [locale.members]

```
template <class Facet> locale combine(const locale& other) const;
```

1 *Effects:* Constructs a locale incorporating all facets from `*this` except for that one facet of `other` that is identified by `Facet`.

2 *Returns:* The newly created locale.

3 *Throws:* `runtime_error` if `has_facet<Facet>(other)` is false.

4 *Remarks:* The resulting locale has no name.

```
basic_string<char> name() const;
```

5 *Returns:* The name of `*this`, if it has one; otherwise, the string `"*"`. If `*this` has a name, then `locale(name().c_str())` is equivalent to `*this`. Details of the contents of the resulting string are otherwise implementation-defined.

22.1.1.4 `locale` operators [locale.operators]

```
bool operator==(const locale& other) const;
```

1 *Returns:* `true` if both arguments are the same locale, or one is a copy of the other, or each has a name and the names are identical; `false` otherwise.

```
bool operator!=(const locale& other) const;
```

2 *Returns:* The result of the expression: `!(*this == other)`.

```
template <class charT, class Traits, class Allocator>
  bool operator()(const basic_string<charT,Traits,Allocator>& s1,
                  const basic_string<charT,Traits,Allocator>& s2) const;
```

3 *Effects:* Compares two strings according to the `collate<charT>` facet.

4 *Remarks:* This member operator template (and therefore `locale` itself) satisfies requirements for a comparator predicate template argument (Clause 25) applied to strings.

5 *Returns:* The result of the following expression:

```
use_facet< collate<charT> >(*this).compare
  (s1.data(), s1.data()+s1.size(), s2.data(), s2.data()+s2.size()) < 0;
```

6 [*Example:* A vector of strings `v` can be collated according to collation rules in locale `loc` simply by (25.3.1, 23.2.6):

```
std::sort(v.begin(), v.end(), loc);
```

— *end example*]

22.1.1.5 `locale` static members [locale.statics]

```
static locale global(const locale& loc);
```

1 Sets the global locale to its argument.

2 *Effects:* Causes future calls to the constructor `locale()` to return a copy of the argument. If the argument has a name, does

```
std::setlocale(LC_ALL, loc.name().c_str());
```

otherwise, the effect on the C locale, if any, is implementation-defined. No library function other than `locale::global()` shall affect the value returned by `locale()`.

3 *Returns:* The previous value of `locale()`.

```
static const locale& classic();
```

4 The `"C"` locale.

5 *Returns:* A locale that implements the classic `"C"` locale semantics, equivalent to the value `locale("C")`.

6 *Remarks:* This locale, its facets, and their member functions, do not change with time.

22.1.2 `locale` globals [locale.global.templates]

```
template <class Facet> const Facet& use_facet(const locale& loc);
```

1 *Requires:* `Facet` is a facet class whose definition contains the public static member `id` as defined in 22.1.1.1.2.

2 *Returns:* a reference to the corresponding facet of `loc`, if present.

3 *Throws:* `bad_cast` if `has_facet<Facet>(loc)` is `false`.

4 *Remarks:* The reference returned remains valid at least as long as any copy of `loc` exists.

```
template <class Facet> bool has_facet(const locale& loc) throw();
```

5 *Returns:* true if the facet requested is present in `loc`; otherwise false.

22.1.3 Convenience interfaces [locale.convenience]

22.1.3.1 Character classification [classification]

```
template <class charT> bool isspace (charT c, const locale& loc);
template <class charT> bool isprint (charT c, const locale& loc);
template <class charT> bool iscntrl (charT c, const locale& loc);
template <class charT> bool isupper (charT c, const locale& loc);
template <class charT> bool islower (charT c, const locale& loc);
template <class charT> bool isalpha (charT c, const locale& loc);
```

```
template <class charT> bool isdigit (charT c, const locale& loc);
template <class charT> bool ispunct (charT c, const locale& loc);
template <class charT> bool isxdigit(charT c, const locale& loc);
template <class charT> bool isalnum (charT c, const locale& loc);
template <class charT> bool isgraph (charT c, const locale& loc);
```

1 Each of these functions `is`***F*** returns the result of the expression:

```
use_facet< ctype<charT> >(loc).is(ctype_base::F, c)
```

where ***F*** is the `ctype_base::mask` value corresponding to that function (22.2.1).[234]

22.1.3.2 Conversions [conversions]

22.1.3.2.1 Character conversions [conversions.character]

```
template <class charT> charT toupper(charT c, const locale& loc);
```

1 *Returns:* `use_facet<ctype<charT> >(loc).toupper(c)`.

```
template <class charT> charT tolower(charT c, const locale& loc);
```

2 *Returns:* `use_facet<ctype<charT> >(loc).tolower(c)`.

22.1.3.2.2 `string` conversions [conversions.string]

1 Class template `wstring_convert` performs conversions between a wide string and a byte string. It lets you specify a code conversion facet (like class template `codecvt`) to perform the conversions, without affecting any streams or locales. [*Example:* Say, for example, you have a code conversion facet called `codecvt_utf8` that you want to use to output to `cout` a UTF-8 multibyte sequence corresponding to a wide string, but you don't want to alter the locale for `cout`. You can write something like:

```
wstring_convert<codecvt_utf8<wchar_t>> myconv;
std::string mbstring = myconv.to_bytes(L"Hello\n");
std::cout << mbstring;
```

— *end example*]

2 **Class template `wstring_convert` synopsis**

```
namespace std {
template<class Codecvt, class Elem = wchar_t> class wstring_convert {
  public:
    typedef std::basic_string<char> byte_string;
    typedef std::basic_string<Elem> wide_string;
    typedef typename Codecvt::state_type state_type;
    typedef typename wide_string::traits_type::int_type int_type;

    wstring_convert(Codecvt *pcvt = new Codecvt);
    wstring_convert(Codecvt *pcvt, state_type state);
    wstring_convert(const byte_string& byte_err,
                    const wide_string& wide_err = wide_string());
    ~wstring_convert();

    wide_string from_bytes(char byte);
    wide_string from_bytes(const char *ptr);
```

234) When used in a loop, it is faster to cache the `ctype<>` facet and use it directly, or use the vector form of `ctype<>::is`.

```
    wide_string from_bytes(const byte_string& str);
    wide_string from_bytes(const char *first, const char *last);

    byte_string to_bytes(Elem wchar);
    byte_string to_bytes(const Elem *wptr);
    byte_string to_bytes(const wide_string& wstr);
    byte_string to_bytes(const Elem *first, const Elem *last);

    size_t converted() const;
    state_type state() const;
  private:
    byte_string byte_err_string;    // exposition only
    wide_string wide_err_string;    // exposition only
    Codecvt *cvtptr;                // exposition only
    state_type cvtstate;            // exposition only
    size_t cvtcount;                // exposition only
  };
}
```

3 The class template describes an object that controls conversions between wide string objects of class `std::basic_string<Elem>` and byte string objects of class `std::basic_string<char>` (also known as `std::string`). The class template defines the types `wide_string` and `byte_string` as synonyms for these two types. Conversion between a sequence of `Elem` values (stored in a wide_string object) and multibyte sequences (stored in a byte_string object) is performed by an object of class `Codecvt<Elem, char, std::mbstate_t>`, which meets the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

4 An object of this class template stores:

— `byte_err_string` — a byte string to display on errors

— `wide_err_string` — a wide string to display on errors

— `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wstring_convert` object is destroyed)

— `cvtstate` — a conversion state object

— `cvtcount` — a conversion count

```
typedef std::basic_string<char> byte_string;
```

5 The type shall be a synonym for `std::basic_string<char>`

```
size_t converted() const;
```

6 *Returns:* `cvtcount`.

```
wide_string from_bytes(char byte);
wide_string from_bytes(const char *ptr);
wide_string from_bytes(const byte_string& str);
wide_string from_bytes(const char *first, const char *last);
```

7 *Effects:* The first member function shall convert the single-element sequence `byte` to a wide string. The second member function shall convert the nul-terminated sequence beginning at `ptr` to a wide string. The third member function shall convert the sequence stored in `str` to a wide string. The fourth member function shall convert the sequence defined by the range `[first,last)` to a wide string.

8 In all cases:

— If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

— The number of input elements successfully converted shall be stored in `cvtcount`.

9 *Returns:* If no conversion error occurs, the member function shall return the converted wide string. Otherwise, if the object was constructed with a wide-error string, the member function shall return the wide-error string. Otherwise, the member function throws an object of class `std::range_error`.

```
typedef typename wide_string::traits_type::int_type int_type;
```

The type shall be a synonym for `wide_string::traits_type::int_type`.

```
state_type state() const;
```

10 returns `cvtstate`.

```
typedef typename Codecvt::state_type state_type;
```

11 The type shall be a synonym for `Codecvt::state_type`.

```
byte_string to_bytes(Elem wchar);
byte_string to_bytes(const Elem *wptr);
byte_string to_bytes(const wide_string& wstr);
byte_string to_bytes(const Elem *first, const Elem *last);
```

12 *Effects:* The first member function shall convert the single-element sequence `wchar` to a byte string. The second member function shall convert the nul-terminated sequence beginning at `wptr` to a byte string. The third member function shall convert the sequence stored in `wstr` to a byte string. The fourth member function shall convert the sequence defined by the range `[first,last)` to a byte string.

13 In all cases:

— If the `cvtstate` object was not constructed with an explicit value, it shall be set to its default value (the initial conversion state) before the conversion begins. Otherwise it shall be left unchanged.

— The number of input elements successfully converted shall be stored in `cvtcount`.

14 *Returns:* If no conversion error occurs, the member function shall return the converted byte string. Otherwise, if the object was constructed with a byte-error string, the member function shall return the byte-error string. Otherwise, the member function shall throw an object of class `std::range_error`.

```
typedef std::basic_string<Elem> wide_string;
```

15 The type shall be a synonym for `std::basic_string<Elem>`.

```
wstring_convert(Codecvt *pcvt = new Codecvt);
wstring_convert(Codecvt *pcvt, state_type state);
wstring_convert(const byte_string& byte_err,
    const wide_string& wide_err = wide_string());
```

16 *Effects:* The first constuctor shall store `pcvt` in `cvtptr` and default values in `cvtstate`, `byte_err_string`, and `wide_err_string`. The second constructor shall store `pcvt` in `cvtptr`, `state` in `cvtstate`, and default values in `byte_err_string` and `wide_err_string`; moreover the stored state shall be retained between calls to `from_bytes` and `to_bytes`. The third constructor shall store `new Codecvt` in `cvtptr`, `state_type()` in `cvtstate`, `byte_err` in `byte_err_string`, and `wide_err` in `wide_err_string`.

```
~wstring_convert();
```

17 *Effects:* The destructor shall delete `cvtptr`.

22.1.3.2.3 Buffer conversions [conversions.buffer]

1 Class template `wbuffer_convert` looks like a wide stream buffer, but performs all its I/O through an underlying byte stream buffer that you specify when you construct it. Like class template `wstring_convert`, it lets you specify a code conversion facet to perform the conversions, without affecting any streams or locales.

2 **Class template `wbuffer_convert` synopsis**

```
namespace std {
template<class Codecvt,
    class Elem = wchar_t,
    class Tr = std::char_traits<Elem> >
  class wbuffer_convert
    : public std::basic_streambuf<Elem, Tr> {
  public:
    typedef typename Tr::state_type state_type;

    wbuffer_convert(std::streambuf *bytebuf = 0,
                    Codecvt *pcvt = new Codecvt,
                    state_type state = state_type());

    std::streambuf *rdbuf() const;
    std::streambuf *rdbuf(std::streambuf *bytebuf);

    state_type state() const;

  private:
    std::streambuf *bufptr;          // exposition only
    Codecvt *cvtptr;                 // exposition only
    state_type cvtstate;             // exposition only
    };
}
```

3 The class template describes a stream buffer that controls the transmission of elements of type `Elem`, whose character traits are described by the class `Tr`, to and from a byte stream buffer of type `std::streambuf`. Conversion between a sequence of `Elem` values and multibyte sequences is performed by an object of class `Codecvt<Elem, char, std::mbstate_t>`, which shall meet the requirements of the standard code-conversion facet `std::codecvt<Elem, char, std::mbstate_t>`.

4 An object of this class template stores:

- `bufptr` — a pointer to its underlying byte stream buffer
- `cvtptr` — a pointer to the allocated conversion object (which is freed when the `wbuffer_convert` object is destroyed)
- `cvtstate` — a conversion state object

```
state_type state() const;
```

5 *Returns:* `cvtstate`.

```
std::streambuf *rdbuf() const;
```

6 *Returns:* `bufptr`.

```
std::streambuf *rdbuf(std::streambuf *bytebuf);
```

7 *Effects:* stores `bytebuf` in `bufptr`.

8 *Returns:* the previous value of `bufptr`.

```
typedef typename Codecvt::state_type state_type;
```

9 The type shall be a synonym for `Codecvt::state_type`.

```
wbuffer_convert(std::streambuf *bytebuf = 0,
    Codecvt *pcvt = new Codecvt, state_type state = state_type());
```

10 *Effects:* The constructor constructs a stream buffer object, initializes `bufptr` to `bytebuf`, initializes `cvtptr` to `pcvt`, and initializes `cvtstate` to `state`.

```
~wbuffer_convert();
```

11 *Effects:* The destructor shall delete `cvtptr`.

22.2 Standard `locale` categories [locale.categories]

1 Each of the standard categories includes a family of facets. Some of these implement formatting or parsing of a datum, for use by standard or users' iostream operators `<<` and `>>`, as members `put()` and `get()`, respectively. Each such member function takes an `ios_base&` argument whose members `flags()`, `precision()`, and `width()`, specify the format of the corresponding datum (27.4.2). Those functions which need to use other facets call its member `getloc()` to retrieve the locale imbued there. Formatting facets use the character argument `fill` to fill out the specified width where necessary.

2 The `put()` members make no provision for error reporting. (Any failures of the OutputIterator argument must be extracted from the returned iterator.) The `get()` members take an `ios_base::iostate&` argument whose value they ignore, but set to `ios_base::failbit` in case of a parse error.

22.2.1 The `ctype` category [category.ctype]

```
namespace std {
  class ctype_base {
  public:
    typedef T mask;

    // numeric values are for exposition only.
    static const mask space = 1 << 0;
    static const mask print = 1 << 1;
    static const mask cntrl = 1 << 2;
    static const mask upper = 1 << 3;
    static const mask lower = 1 << 4;
    static const mask alpha = 1 << 5;
    static const mask digit = 1 << 6;
    static const mask punct = 1 << 7;
    static const mask xdigit = 1 << 8;
    static const mask alnum = alpha | digit;
    static const mask graph = alnum | punct;
  };
}
```

1 The type `mask` is a bitmask type (17.5.3.2.3).

22.2.1.1 Class template `ctype` [locale.ctype]

```
namespace std {
  template <class charT>
  class ctype : public locale::facet, public ctype_base {
  public:
    typedef charT char_type;

    explicit ctype(size_t refs = 0);

    bool         is(mask m, charT c) const;
    const charT* is(const charT* low, const charT* high, mask* vec) const;
    const charT* scan_is(mask m,
                         const charT* low, const charT* high) const;
    const charT* scan_not(mask m,
                          const charT* low, const charT* high) const;
    charT        toupper(charT c) const;
    const charT* toupper(charT* low, const charT* high) const;
    charT        tolower(charT c) const;
    const charT* tolower(charT* low, const charT* high) const;

    charT        widen(char c) const;
    const char*  widen(const char* low, const char* high, charT* to) const;
    char         narrow(charT c, char dfault) const;
    const charT* narrow(const charT* low, const charT*, char dfault,
                        char* to) const;

    static locale::id id;

  protected:
   ~ctype();
    virtual bool         do_is(mask m, charT c) const;
    virtual const charT* do_is(const charT* low, const charT* high,
                               mask* vec) const;
    virtual const charT* do_scan_is(mask m,
                                    const charT* low, const charT* high) const;
    virtual const charT* do_scan_not(mask m,
                                     const charT* low, const charT* high) const;
    virtual charT        do_toupper(charT) const;
    virtual const charT* do_toupper(charT* low, const charT* high) const;
    virtual charT        do_tolower(charT) const;
    virtual const charT* do_tolower(charT* low, const charT* high) const;
    virtual charT        do_widen(char) const;
    virtual const char*  do_widen(const char* low, const char* high,
                                  charT* dest) const;
    virtual char         do_narrow(charT, char dfault) const;
    virtual const charT* do_narrow(const charT* low, const charT* high,
                                   char dfault, char* dest) const;
  };
}
```

1 Class `ctype` encapsulates the C library `<cctype>` features. `istream` members are required to use `ctype<>` for character classing during input parsing.

2 The specializations required in Table 66 (22.1.1.1.1), namely `ctype<char>` and `ctype<wchar_t>`, implement character classing appropriate to the implementation's native character set.

22.2.1.1.1 `ctype` members [locale.ctype.members]

```
bool         is(mask m, charT c) const;
const charT* is(const charT* low, const charT* high,
                mask* vec) const;
```

1 *Returns:* `do_is(m,c)` or `do_is(low,high,vec)`

```
const charT* scan_is(mask m,
                     const charT* low, const charT* high) const;
```

2 *Returns:* `do_scan_is(m,low,high)`

```
const charT* scan_not(mask m,
                      const charT* low, const charT* high) const;
```

3 *Returns:* `do_scan_not(m,low,high)`

```
charT        toupper(charT) const;
const charT* toupper(charT* low, const charT* high) const;
```

4 *Returns:* `do_toupper(c)` or `do_toupper(low,high)`

```
charT        tolower(charT c) const;
const charT* tolower(charT* low, const charT* high) const;
```

5 *Returns:* `do_tolower(c)` or `do_tolower(low,high)`

```
charT       widen(char c) const;
const char* widen(const char* low, const char* high, charT* to) const;
```

6 *Returns:* `do_widen(c)` or `do_widen(low,high,to)`

```
char         narrow(charT c, char dfault) const;
const charT* narrow(const charT* low, const charT*, char dfault,
                    char* to) const;
```

7 *Returns:* `do_narrow(c,dfault)` or `do_narrow(low,high,dfault,to)`

22.2.1.1.2 `ctype` virtual functions [locale.ctype.virtuals]

```
bool         do_is(mask m, charT c) const;
const charT* do_is(const charT* low, const charT* high,
                   mask* vec) const;
```

1 *Effects:* Classifies a character or sequence of characters. For each argument character, identifies a value `M` of type `ctype_base::mask`. The second form identifies a value `M` of type `ctype_base::mask` for each `*p` where `(low<=p && p<high)`, and places it into `vec[p-low]`.

2 *Returns:* The first form returns the result of the expression `(M & m) != 0`; i.e., `true` if the character has the characteristics specified. The second form returns `high`.

```
const charT* do_scan_is(mask m,
                        const charT* low, const charT* high) const;
```

3 *Effects:* Locates a character in a buffer that conforms to a classification `m`.

4 *Returns:* The smallest pointer `p` in the range `[low, high)` such that `is(m,*p)` would return `true`; otherwise, returns `high`.

```
const charT* do_scan_not(mask m,
                         const charT* low, const charT* high) const;
```

5 *Effects:* Locates a character in a buffer that fails to conform to a classification `m`.

6 *Returns:* The smallest pointer `p`, if any, in the range `[low,high)` such that `is(m,*p)` would return `false`; otherwise, returns `high`.

```
charT        do_toupper(charT c) const;
const charT* do_toupper(charT* low, const charT* high) const;
```

7 *Effects:* Converts a character or characters to upper case. The second form replaces each character `*p` in the range `[low,high)` for which a corresponding upper-case character exists, with that character.

8 *Returns:* The first form returns the corresponding upper-case character if it is known to exist, or its argument if not. The second form returns `high`.

```
charT        do_tolower(charT c) const;
const charT* do_tolower(charT* low, const charT* high) const;
```

9 *Effects:* Converts a character or characters to lower case. The second form replaces each character `*p` in the range `[low,high)` and for which a corresponding lower-case character exists, with that character.

10 *Returns:* The first form returns the corresponding lower-case character if it is known to exist, or its argument if not. The second form returns `high`.

```
charT        do_widen(char c) const;
const char*  do_widen(const char* low, const char* high,
                      charT* dest) const;
```

11 *Effects:* Applies the simplest reasonable transformation from a `char` value or sequence of `char` values to the corresponding `charT` value or values.[235] The only characters for which unique transformations are required are those in the basic source character set (2.2).

For any named `ctype` category with a `ctype<charT>` facet `ctc` and valid `ctype_base::mask` value `M`, `(ctc.is(M, c) || !is(M, do_widen(c)) )` is `true`.[236]

The second form transforms each character `*p` in the range `[low,high)`, placing the result in `dest[p-low]`.

12 *Returns:* The first form returns the transformed value. The second form returns `high`.

```
char         do_narrow(charT c, char dfault) const;
const charT* do_narrow(const charT* low, const charT* high,
                       char dfault, char* dest) const;
```

13 *Effects:* Applies the simplest reasonable transformation from a `charT` value or sequence of `charT` values to the corresponding `char` value or values.

For any character `c` in the basic source character set (2.2) the transformation is such that

```
do_widen(do_narrow(c,0)) == c
```

For any named `ctype` category with a `ctype<char>` facet `ctc` however, and `ctype_base::mask` value `M`,

235) The char argument of `do_widen` is intended to accept values derived from character literals for conversion to the locale's encoding.

236) In other words, the transformed character is not a member of any character classification that `c` is not also a member of.

```
(is(M,c) || !ctc.is(M, do_narrow(c,dfault)) )
```

is `true` (unless `do_narrow` returns `dfault`). In addition, for any digit character `c`, the expression `(do_narrow(c, dfault) - '0')` evaluates to the digit value of the character. The second form transforms each character `*p` in the range `[low,high)`, placing the result (or `dfault` if no simple transformation is readily available) in `dest[p-low]`.

14 *Returns:* The first form returns the transformed value; or `dfault` if no mapping is readily available. The second form returns `high`.

22.2.1.2 Class template `ctype_byname` [locale.ctype.byname]

```
namespace std {
  template <class charT>
  class ctype_byname : public ctype<charT> {
  public:
    typedef typename ctype<charT>::mask mask;
    explicit ctype_byname(const char*, size_t refs = 0);
    explicit ctype_byname(const string&, size_t refs = 0);
  protected:
   ~ctype_byname();
  };
}
```

22.2.1.3 `ctype` specializations [facet.ctype.special]

```
namespace std {
  template <> class ctype<char>
    : public locale::facet, public ctype_base {
  public:
    typedef char char_type;

    explicit ctype(const mask* tab = 0, bool del = false,
                   size_t refs = 0);

    bool is(mask m, char c) const;
    const char* is(const char* low, const char* high, mask* vec) const;
    const char* scan_is (mask m,
                         const char* low, const char* high) const;
    const char* scan_not(mask m,
                         const char* low, const char* high) const;

    char        toupper(char c) const;
    const char* toupper(char* low, const char* high) const;
    char        tolower(char c) const;
    const char* tolower(char* low, const char* high) const;

    char  widen(char c) const;
    const char* widen(const char* low, const char* high, char* to) const;
    char  narrow(char c, char dfault) const;
    const char* narrow(const char* low, const char* high, char dfault,
                       char* to) const;

    static locale::id id;
    static const size_t table_size = implementation-defined;
```

```
    const mask* table() const throw();
    static const mask* classic_table() throw();

  protected:
   ~ctype();
    virtual char        do_toupper(char c) const;
    virtual const char* do_toupper(char* low, const char* high) const;
    virtual char        do_tolower(char c) const;
    virtual const char* do_tolower(char* low, const char* high) const;

    virtual char        do_widen(char c) const;
    virtual const char* do_widen(const char* low,
                                 const char* high,
                                 char* to) const;
    virtual char        do_narrow(char c, char dfault) const;
    virtual const char* do_narrow(const char* low,
                                  const char* high,
                                  char dfault, char* to) const;
  };
}
```

1 A specialization `ctype<char>` is provided so that the member functions on type `char` can be implemented `inline`.[237] The implementation-defined value of member `table_size` is at least 256.

22.2.1.3.1 `ctype<char>` destructor [facet.ctype.char.dtor]

```
~ctype();
```

1 *Effects:* If the constructor's first argument was nonzero, and its second argument was true, does `delete [] table()`.

22.2.1.3.2 `ctype<char>` members [facet.ctype.char.members]

1 In the following member descriptions, for `unsigned char` values `v` where `(v >= table_size)`, `table()[v]` is assumed to have an implementation-defined value (possibly different for each such value `v`) without performing the array lookup.

```
explicit ctype(const mask* tbl = 0, bool del = false,
               size_t refs = 0);
```

2 *Precondition:* `tbl` either 0 or an array of at least `table_size` elements.

3 *Effects:* Passes its `refs` argument to its base class constructor.

```
bool        is(mask m, char c) const;
const char* is(const char* low, const char* high,
               mask* vec) const;
```

4 *Effects:* The second form, for all `*p` in the range `[low,high)`, assigns into `vec[p-low]` the value `table()[(unsigned char)*p]`.

5 *Returns:* The first form returns `table()[(unsigned char)c] & m`; the second form returns `high`.

237) Only the `char` (not `unsigned char` and `signed char`) form is provided. The specialization is specified in the standard, and not left as an implementation detail, because it affects the derivation interface for `ctype<char>`.

```
const char* scan_is(mask m,
                    const char* low, const char* high) const;
```

6 *Returns:* The smallest `p` in the range `[low,high)` such that

```
table()[(unsigned char) *p] & m
```

is `true`.

```
const char* scan_not(mask m,
                     const char* low, const char* high) const;
```

7 *Returns:* The smallest `p` in the range `[low,high)` such that

```
table()[(unsigned char) *p] & m
```

is `false`.

```
char        toupper(char c) const;
const char* toupper(char* low, const char* high) const;
```

8 *Returns:* `do_toupper(c)` or `do_toupper(low,high)`, respectively.

```
char        tolower(char c) const;
const char* tolower(char* low, const char* high) const;
```

9 *Returns:* `do_tolower(c)` or `do_tolower(low,high)`, respectively.

```
char  widen(char c) const;
const char* widen(const char* low, const char* high,
    char* to) const;
```

10 *Returns:* `do_widen(c)` or `do_widen(low, high, to)`, respectively.

```
char        narrow(char c, char dfault) const;
const char* narrow(const char* low, const char* high,
                   char dfault, char* to) const;
```

11 *Returns:* `do_narrow(c, dfault)` or `do_narrow(low, high, dfault, to)`, respectively.

```
const mask* table() const throw();
```

12 *Returns:* The first constructor argument, if it was non-zero, otherwise `classic_table()`.

22.2.1.3.3 `ctype<char>` static members [facet.ctype.char.statics]

```
static const mask* classic_table() throw();
```

1 *Returns:* A pointer to the initial element of an array of size `table_size` which represents the classifications of characters in the "C" locale.

22.2.1.3.4 `ctype<char>` virtual functions [facet.ctype.char.virtuals]

```
  char        do_toupper(char) const;
  const char* do_toupper(char* low, const char* high) const;
  char        do_tolower(char) const;
  const char* do_tolower(char* low, const char* high) const;

  virtual char        do_widen(char c) const;
```

```
virtual const char* do_widen(const char* low,
                             const char* high,
                             char* to) const;
virtual char        do_narrow(char c, char dfault) const;
virtual const char* do_narrow(const char* low,
                              const char* high,
                              char dfault, char* to) const;
```

These functions are described identically as those members of the same name in the `ctype` class template (22.2.1.1.1).

22.2.1.4 Class template codecvt [locale.codecvt]

```
namespace std {
  class codecvt_base {
  public:
    enum result { ok, partial, error, noconv };
  };

  template <class internT, class externT, class stateT>
  class codecvt : public locale::facet, public codecvt_base {
  public:
    typedef internT  intern_type;
    typedef externT  extern_type;
    typedef stateT state_type;

    explicit codecvt(size_t refs = 0);

    result out(stateT& state,
               const internT* from, const internT* from_end, const internT*& from_next,
               externT*   to,       externT* to_limit, externT*& to_next) const;
    result unshift(stateT& state,
                   externT*   to,        externT* to_limit, externT*& to_next) const;
    result in(stateT& state,
              const externT* from, const externT* from_end, const externT*& from_next,
              internT*   to,       internT* to_limit, internT*& to_next) const;
    int encoding() const throw();
    bool always_noconv() const throw();
    int length(stateT&, const externT* from, const externT* end,
               size_t max) const;
    int max_length() const throw();

    static locale::id id;

  protected:
    ~codecvt();
    virtual result do_out(stateT& state,
                          const internT* from, const internT* from_end, const internT*& from_next,
                          externT* to,         externT* to_limit, externT*& to_next) const;
    virtual result do_in(stateT& state,
                         const externT* from, const externT* from_end, const externT*& from_next,
                         internT* to,         internT* to_limit, internT*& to_next) const;
    virtual result do_unshift(stateT& state,
                              externT* to,         externT* to_limit, externT*& to_next) const;
    virtual int do_encoding() const throw();
    virtual bool do_always_noconv() const throw();
```

```
    virtual int do_length(stateT&, const externT* from,
                          const externT* end, size_t max) const;
    virtual int do_max_length() const throw();
  };
}
```

1 The class `codecvt<internT,externT,stateT>` is for use when converting from one codeset to another, such as from wide characters to multibyte characters or between wide character encodings such as Unicode and EUC.

2 The `stateT` argument selects the pair of codesets being mapped between.

3 The specializations required in Table 66 (22.1.1.1.1) convert the implementation-defined native character set. `codecvt<char, char, mbstate_t>` implements a degenerate conversion; it does not convert at all. The specialization `codecvt<char16_t, char, mbstate_t>` converts between the UTF-16 and UTF-8 encodings schemes, and the specialization `codecvt <char32_t, char, mbstate_t>` converts between the UTF-32 and UTF-8 encodings schemes. `codecvt<wchar_t,char,mbstate_t>` converts between the native character sets for narrow and wide characters. Specializations on `mbstate_t` perform conversion between encodings known to the library implementor. Other encodings can be converted by specializing on a user-defined `stateT` type. The `stateT` object can contain any state that is useful to communicate to or from the specialized `do_in` or `do_out` members.

22.2.1.4.1 `codecvt` members [locale.codecvt.members]

```
result out(stateT& state,
  const internT* from, const internT* from_end, const internT*& from_next,
        externT* to, externT* to_limit, externT*& to_next) const;
```

1 *Returns:* `do_out(state, from, from_end, from_next, to,to_limit, to_next)`

```
result unshift(stateT& state,
        externT* to, externT* to_limit, externT*& to_next) const;
```

2 *Returns:* `do_unshift(state, to, to_limit, to_next)`

```
result in(stateT& state,
  const externT* from, const externT* from_end, const externT*& from_next,
        internT* to, internT* to_limit, internT*& to_next) const;
```

3 *Returns:* `do_in(state, from,from_end,from_next, to,to_limit,to_next)`

```
int encoding() const throw();
```

4 *Returns:* `do_encoding()`

```
bool always_noconv() const throw();
```

5 *Returns:* `do_always_noconv()`

```
int length(stateT& state, const externT* from, const externT* from_end,
           size_t max) const;
```

6 *Returns:* `do_length(state, from,from_end,max)`

```
int max_length() const throw();
```

7 *Returns:* `do_max_length()`

22.2.1.4.2 codecvt virtual functions [locale.codecvt.virtuals]

```
result do_out(stateT& state,
  const internT* from, const internT* from_end, const internT*& from_next,
  externT* to, externT* to_limit, externT*& to_next) const;

result do_in(stateT& state,
  const externT* from, const externT* from_end, const externT*& from_next,
        internT* to, internT* to_limit, internT*& to_next) const;
```

1 *Preconditions:* `(from<=from_end && to<=to_end)` well-defined and `true`; `state` initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

2 *Effects:* Translates characters in the source range `[from,from_end)`, placing the results in sequential positions starting at destination `to`. Converts no more than `(from_end-from)` source elements, and stores no more than `(to_limit-to)` destination elements.

Stops if it encounters a character it cannot convert. It always leaves the `from_next` and `to_next` pointers pointing one beyond the last element successfully converted. If returns `noconv`, `internT` and `externT` are the same type and the converted sequence is identical to the input sequence `[from, from_next)`. `to_next` is set equal to `to`, the value of `state` is unchanged, and there are no changes to the values in `[to, to_limit)`.

3 A `codecvt` facet that is used by `basic_filebuf` (27.8) shall have the property that if

```
do_out(state, from, from_end, from_next, to, to_limit, to_next)
```

would return `ok`, where `from != from_end`, then

```
do_out(state, from, from + 1, from_next, to, to_end, to_next)
```

shall also return `ok`, and that if

```
do_in(state, from, from_end, from_next, to, to_limit, to_next)
```

would return `ok`, where `to != to_end`, then

```
do_in(state, from, from_end, from_next, to, to + 1, to_next)
```

shall also return `ok`.[238] [*Note:* As a result of operations on `state`, it can return `ok` or `partial` and set `next == from` and `to_next != to`. — *end note*]

4 *Remarks:* Its operations on `state` are unspecified. [*Note:* This argument can be used, for example, to maintain shift state, to specify conversion options (such as count only), or to identify a cache of seek offsets. — *end note*]

5 *Returns:* An enumeration value, as summarized in Table 68.

A return value of `partial`, if `(from_next==from_end)`, indicates that either the destination sequence has not absorbed all the available destination elements, or that additional source elements are needed before another destination element can be produced.

```
result do_unshift(stateT& state,
  externT* to, externT* to_limit, externT*& to_next) const;
```

238) Informally, this means that `basic_filebuf` assumes that the mappings from internal to external characters is 1 to N: a `codecvt` facet that is used by `basic_filebuf` must be able to translate characters one internal character at a time.

Table 68 — `do_in/do_out` result values

Value	Meaning
`ok`	completed the conversion
`partial`	not all source characters converted
`error`	encountered a character in `[from,from_end)` that it could not convert
`noconv`	`internT` and `externT` are the same type, and input sequence is identical to converted sequence

6 *Requires:* `(to <= to_end)` well defined and true; state initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

7 *Effects:* Places characters starting at `to` that should be appended to terminate a sequence when the current `stateT` is given by `state`.[239] Stores no more than `(to_limit-to)` destination elements, and leaves the `to_next` pointer pointing one beyond the last element successfully stored.

8 *Returns:* An enumeration value, as summarized in Table 69.

Table 69 — `do_unshift` result values

Value	Meaning
`ok`	completed the sequence
`partial`	space for more than `to_limit-to` destination elements was needed to terminate a sequence given the value of `state`
`error`	an unspecified error has occurred
`noconv`	no termination is needed for this `state_type`

```
int do_encoding() const throw();
```

9 *Returns:*-1 if the encoding of the externT sequence is state-dependent; else the constant number of externT characters needed to produce an internal character; or 0 if this number is not a constant[240].

```
bool do_always_noconv() const throw();
```

10 *Returns:* `true` if `do_in()` and `do_out()` return `noconv` for all valid argument values. `codecvt<char, char, mbstate_t>` returns `true`.

```
int do_length(stateT& state, const externT* from, const externT* from_end,
              size_t max) const;
```

11 *Preconditions:* `(from<=from_end)` well-defined and `true`; `state` initialized, if at the beginning of a sequence, or else equal to the result of converting the preceding characters in the sequence.

12 *Effects:* The effect on the `state` argument is "as if" it called `do_in(state, from, from_end, from, to, to+max, to)` for `to` pointing to a buffer of at least `max` elements.

13 *Returns:* `(from_next-from)` where `from_next` is the largest value in the range `[from,from_end]` such that the sequence of values in the range `[from,from_next)` represents `max` or fewer valid complete

239) Typically these will be characters to return the state to `stateT()`

240) If encoding() yields -1, then more than max_length() externT elements may be consumed when producing a single internT character, and additional externT elements may appear at the end of a sequence after those that yield the final internT character.

characters of type `internT`. The specialization `codecvt<char, char, mbstate_t>`, returns the lesser of `max` and `(from_end-from)`.

```
int do_max_length() const throw();
```

14 *Returns:* The maximum value that `do_length(state, from, from_end, 1)` can return for any valid range `[from, from_end)` and `stateT` value `state`. The specialization `codecvt<char, char, mbstate_t>::do_max_length()` returns 1.

22.2.1.5 Class template `codecvt_byname` [locale.codecvt.byname]

```
namespace std {
  template <class internT, class externT, class stateT>
  class codecvt_byname : public codecvt<internT, externT, stateT> {
  public:
    explicit codecvt_byname(const char*, size_t refs = 0);
    explicit codecvt_byname(const string&, size_t refs = 0);
  protected:
    ~codecvt_byname();
  };
}
```

22.2.2 The numeric category [category.numeric]

1 The classes `num_get<>` and `num_put<>` handle numeric formatting and parsing. Virtual functions are provided for several numeric types. Implementations may (but are not required to) delegate extraction of smaller types to extractors for larger types.[241]

2 All specifications of member functions for `num_put` and `num_get` in the subclauses of 22.2.2 only apply to the specializations required in Tables 66 and 67 (22.1.1.1.1), namely `num_get<char>`, `num_get<wchar_t>`, `num_get<C, InputIterator>`, `num_put<char>`, `num_put<wchar_t>`, and `num_put<C,OutputIterator>`. These specializations refer to the `ios_base&` argument for formatting specifications (22.2), and to its imbued locale for the `numpunct<>` facet to identify all numeric punctuation preferences, and also for the `ctype<>` facet to perform character classification.

3 Extractor and inserter members of the standard iostreams use `num_get<>` and `num_put<>` member functions for formatting and parsing numeric values (27.6.1.2.1, 27.6.2.6.1).

22.2.2.1 Class template `num_get` [locale.num.get]

```
namespace std {
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class num_get : public locale::facet {
  public:
    typedef charT            char_type;
    typedef InputIterator    iter_type;

    explicit num_get(size_t refs = 0);

    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, bool& v) const;
    iter_type get(iter_type in, iter_type end, ios_base& ,
                  ios_base::iostate& err, long& v) const;
```

241) Parsing `"-1"` correctly into (e.g.) an `unsigned short` requires that the corresponding member `get()` at least extract the sign before delegating.

```
    iter_type get(iter_type in, iter_type end, ios_base& ,
                  ios_base::iostate& err, long long& v) const;}
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned short& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned int& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, unsigned long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base& ,
                  ios_base::iostate& err, unsigned long long& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, float& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, long double& v) const;
    iter_type get(iter_type in, iter_type end, ios_base&,
                  ios_base::iostate& err, void*& v) const;

    static locale::id id;

  protected:
    ~num_get();
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, bool& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned short& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned int& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, unsigned long long& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, float& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, long double& v) const;
    virtual iter_type do_get(iter_type, iter_type, ios_base&,
                             ios_base::iostate& err, void*& v) const;
  };
}
```

1 The facet num_get is used to parse numeric values from an input sequence such as an istream.

22.2.2.1.1 num_get members [facet.num.get.members]

```
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, bool& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
```

```
  ios_base::iostate& err, long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned short& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned int& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long long& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, float& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long double& val) const;
iter_type get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, void*& val) const;
```

1 *Returns:* `do_get(in, end, str, err, val)`.

22.2.2.1.2 `num_get` virtual functions [facet.num.get.virtuals]

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned short& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned int& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, unsigned long long& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, float& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, long double& val) const;
iter_type do_get(iter_type in, iter_type end, ios_base& str,
  ios_base::iostate& err, void*& val) const;
```

1 *Effects:* Reads characters from `in`, interpreting them according to `str.flags()`, `use_facet<ctype<charT> >(loc)`, and `use_facet< numpunct<charT> >(loc)`, where `loc` is `str.getloc()`. If an error occurs, `val` is unchanged; otherwise it is set to the resulting value.

2 The details of this operation occur in three stages

- Stage 1: Determine a conversion specifier
- Stage 2: Extract characters from `in` and determine a corresponding `char` value for the format expected by the conversion specification determined in stage 1.
- Stage 3: Store results

3 The details of the stages are presented below.

Stage 1: The function initializes local variables via

```
fmtflags flags = str .flags();
fmtflags basefield = (flags & ios_base::basefield);
fmtflags uppercase = (flags & ios_base::uppercase);
fmtflags boolalpha = (flags & ios_base::boolalpha);
```

For conversion to an integral type, the function determines the integral conversion specifier as indicated in Table 70. The table is ordered. That is, the first line whose condition is true applies.

Table 70 — Integer conversions

State	`stdio` equivalent
`basefield == oct`	`%o`
`basefield == hex`	`%X`
`basefield == 0`	`%i`
`signed` integral type	`%d`
`unsigned` integral type	`%u`

For conversions to a floating type the specifier is `%g`.

For conversions to `void*` the specifier is `%p`.

A length modifier is added to the conversion specification, if needed, as indicated in Table 71.

Table 71 — Length modifier

Type	Length modifier
`short`	`h`
`unsigned short`	`h`
`long`	`l`
`unsigned long`	`l`
`long long`	`ll`
`unsigned long long`	`ll`
`double`	`l`
`long double`	`L`

Stage 2: If `in==end` then stage 2 terminates. Otherwise a `charT` is taken from `in` and local variables are initialized as if by

```
char_type ct = *in ;
char c = src[find(atoms, atoms + sizeof(src) - 1, ct) - atoms];
if (ct ==  use_facet<numpunct<charT> >(loc).decimal_point())
c = '.';
bool discard =
  ct == use_facet<numpunct<charT> >(loc).thousands_sep()
  && use_facet<numpunct<charT> >(loc).grouping().length() != 0;
```

where the values `src` and `atoms` are defined as if by:

```
static const char src[] = "0123456789abcdefxABCDEFX+-";
char_type atoms[sizeof(src)];
use_facet<ctype<charT> >(loc).widen(src, src + sizeof(src), atoms);
```

for this value of `loc`.

If `discard` is true, then if `'.'` has not yet been accumulated, then the position of the character is remembered, but the character is otherwise ignored. Otherwise, if `'.'` has already been accumulated, the character is discarded and Stage 2 terminates.

If the character is either discarded or accumulated then `in` is advanced by `++in` and processing returns to the beginning of stage 2.

Stage 3: The sequence of `char`s accumulated in stage 2 (the field) is converted to a numeric value by the rules of one of the functions declared in the header `<cstdlib>`:

- For a signed integer value, the function `strtoll`.
- For an unsigned integer value, the function `strtoull`.
- For a floating-point value, the function `strtold`.

The numeric value to be stored can be one of:

- zero, if the conversion function fails to convert the entire field. `ios_base::failbit` is assigned to `err`.
- the most positive representable value, if the field represents a value too large positive to be represented in `val`. `ios_base::failbit` is assigned to `err`.
- the most negative representable value or zero for an unsigned integer type, if the field represents a value too large negative to be represented in `val`. `ios_base::failbit` is assigned to `err`.
- the converted value, otherwise.

The resultant numeric value is stored in `val`.

4 Digit grouping is checked. That is, the positions of discarded separators is examined for consistency with `use_facet<numpunct<charT> >(loc).grouping()`. If they are not consistent then `ios_base::failbit` is assigned to `err`.

5 In any case, if stage 2 processing was terminated by the test for `in==end` then `err |=ios_base::eofbit` is performed.

```
iter_type do_get(iter_type in, iter_type end, ios_base& str,
                 ios_base::iostate& err, bool& val) const;
```

6 *Effects:* If `(str.flags()&ios_base::boolalpha)==0` then input proceeds as it would for a `long` except that if a value is being stored into `val`, the value is determined according to the following: If the value to be stored is 0 then `false` is stored. If the value is 1 then `true` is stored. Otherwise `true` is stored and `ios_base::failbit` is assigned to `err`.

7 Otherwise target sequences are determined "as if" by calling the members `falsename()` and `truename()` of the facet obtained by `use_facet<numpunct<charT> >(str.getloc())`. Successive characters in the range `[in,end)` (see 23.1.3) are obtained and matched against corresponding positions in the target sequences only as necessary to identify a unique match. The input iterator `in` is compared to `end` only when necessary to obtain a character. If a target sequence is uniquely matched, `val` is set to the corresponding value. Otherwise `false` is stored and `ios_base::failbit` is assigned to `err`.

8 The `in` iterator is always left pointing one position beyond the last character successfully matched. If `val` is set, then `err` is set to `str.goodbit`; or to `str.eofbit` if, when seeking another character to match, it is found that `(in == end)`. If `val` is not set, then `err` is set to `str.failbit`; or to

`(str.failbit|str.eofbit)` if the reason for the failure was that `(in == end)`. [*Example:* For targets `true`: `"a"` and `false`: `"abb"`, the input sequence `"a"` yields `val == true` and `err == str.eofbit`; the input sequence `"abc"` yields `err = str.failbit`, with `in` ending at the `'c'` element. For targets `true`: `"1"` and `false`: `"0"`, the input sequence `"1"` yields `val == true` and `err == str.goodbit`. For empty targets `("")`, any input sequence yields `err == str.failbit`. — *end example*]

9 *Returns:* `in`.

22.2.2.2 Class template num_put [locale.nm.put]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class num_put : public locale::facet {
  public:
    typedef charT            char_type;
    typedef OutputIterator   iter_type;

    explicit num_put(size_t refs = 0);

    iter_type put(iter_type s, ios_base& f, char_type fill, bool v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill, long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  unsigned long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  unsigned long long v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  long double v) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  const void* v) const;

    static locale::id id;

  protected:
    ~num_put();
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             bool v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             long long v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             unsigned long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             unsigned long long) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             double v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             long double v) const;
    virtual iter_type do_put(iter_type, ios_base&, char_type fill,
                             const void* v) const;
  };
}
```

1 The facet num_put is used to format numeric values to a character sequence such as an ostream.

22.2.2.2.1 num_put members [facet.num.put.members]

```
iter_type put(iter_type out, ios_base& str, char_type fill,
  bool val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  unsigned long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  unsigned long long val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  long double val) const;
iter_type put(iter_type out, ios_base& str, char_type fill,
  const void* val) const;
```

1 *Returns:* do_put(out, str, fill, val).

22.2.2.2.2 num_put virtual functions [facet.num.put.virtuals]

```
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  unsigned long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  unsigned long long val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  long double val) const;
iter_type do_put(iter_type out, ios_base& str, char_type fill,
  const void* val) const;
```

1 *Effects:* Writes characters to the sequence out, formatting val as desired. In the following description, a local variable initialized with

```
locale loc = str.getloc();
```

2 The details of this operation occur in several stages:

— Stage 1: Determine a printf conversion specifier spec and determining the characters that would be printed by printf (27.8.2) given this conversion specifier for

```
printf(spec, val )
```

assuming that the current locale is the "C" locale.

— Stage 2: Adjust the representation by converting each char determined by stage 1 to a charT using a conversion and values returned by members of use_facet< numpunct<charT> >(str.getloc())

— Stage 3: Determine where padding is required.

— Stage 4: Insert the sequence into the `out`.

Detailed descriptions of each stage follow.

Returns: `out`.

Stage 1: The first action of stage 1 is to determine a conversion specifier. The tables that describe this determination use the following local variables

```
fmtflags flags = str.flags() ;
fmtflags basefield =  (flags & (ios_base::basefield));
fmtflags uppercase =  (flags & (ios_base::uppercase));
fmtflags floatfield = (flags & (ios_base::floatfield));
fmtflags showpos =    (flags & (ios_base::showpos));
fmtflags showbase =   (flags & (ios_base::showbase));
```

All tables used in describing stage 1 are ordered. That is, the first line whose condition is true applies. A line without a condition is the default behavior when none of the earlier lines apply.

For conversion from an integral type other than a character type, the function determines the integral conversion specifier as indicated in Table 72.

Table 72 — Integer conversions

State	`stdio` equivalent
`basefield == ios_base::oct`	`%o`
`(basefield == ios_base::hex) && !uppercase`	`%x`
`(basefield == ios_base::hex)`	`%X`
for a `signed` integral type	`%d`
for an `unsigned` integral type	`%u`

For conversion from a floating-point type, the function determines the floating-point conversion specifier as indicated in Table 73.

Table 73 — Floating-point conversions

State	`stdio` equivalent
`floatfield == ios_base::fixed`	`%f`
`floatfield == ios_base::scientific && !uppercase`	`%e`
`floatfield == ios_base::scientific`	`%E`
`floatfield == ios_base::fixed | ios_base::scientific` && !uppercase	`%a`
`floatfield == ios_base::fixed | ios_base::scientific`	`%A`
`!uppercase`	`%g`
otherwise	`%G`

For conversions from an integral or floating type a length modifier is added to the conversion specifier as indicated in Table 74.

The conversion specifier has the following optional additional qualifiers prepended as indicated in Table 75.

Table 74 — Length modifier

Type	Length modifier
`long`	`l`
`long long`	`ll`
`unsigned long`	`l`
`unsigned long long`	`ll`
`long double`	`L`
otherwise	*none*

Table 75 — Numeric conversions

Type(s)	State	`stdio` equivalent
an integral type	`flags & showpos`	`+`
	`flags & showbase`	`#`
a floating-point type	`flags & showpos`	`+`
	`flags & showpoint`	`#`

For conversion from a floating-point type, `str.precision()` is specified in the conversion specification.

For conversion from `void*` the specifier is `%p`.

The representations at the end of stage 1 consists of the `char`'s that would be printed by a call of `printf(s, val)` where `s` is the conversion specifier determined above.

Stage 2: Any character `c` other than a decimal point(.) is converted to a `charT` via `use_facet<ctype<charT> >(loc).widen( c )`

A local variable `punct` is initialized via

```
const numpunct<charT>& punct = use_facet< numpunct<charT> >(str.getloc());
```

For arithmetic types, `punct.thousands_sep()` characters are inserted into the sequence as determined by the value returned by `punct.do_grouping()` using the method described in 22.2.3.1.2

Decimal point characters(.) are replaced by `punct.decimal_point()`

Stage 3: A local variable is initialized as

```
fmtflags adjustfield=   (flags & (ios_base::adjustfield));
```

The location of any padding[242] is determined according to Table 76.

If `str.width()` is nonzero and the number of `charT`'s in the sequence after stage 2 is less than `str.width()`, then enough `fill` characters are added to the sequence at the position indicated for padding to bring the length of the sequence to `str.width()`.

`str.width(0)` is called.

Stage 4: The sequence of `charT`'s at the end of stage 3 are output via

```
*out++ = c
```

242) The conversion specification `#o` generates a leading `0` which is *not* a padding character.

Table 76 — Fill padding

State	Location
`adjustfield == ios_base::left`	pad after
`adjustfield == ios_base::right`	pad before
`adjustfield == internal` and a sign occurs in the representation	pad after the sign
`adjustfield == internal` and representation after stage 1 began with 0x or 0X	pad after x or X
otherwise	pad before

```
iter_type do_put(iter_type out, ios_base& str, char_type fill,
                 bool val) const;
```

6 *Returns:* If `(str.flags() & ios_base::boolalpha) == 0` returns `do_put(out, str, fill, (int)val)`, otherwise obtains a string `s` as if by

```
string_type s =
  val ? use_facet<ctype<charT> >(loc).truename()
    : use_facet<ctype<charT> >(loc).falsename();
```

and then inserts each character `c` of `s` into `out` via `*out++ = c` and returns `out`.

22.2.3 The numeric punctuation facet [facet.numpunct]

22.2.3.1 Class template `numpunct` [locale.numpunct]

```
namespace std {
  template <class charT>
  class numpunct : public locale::facet {
  public:
    typedef charT               char_type;
    typedef basic_string<charT> string_type;

    explicit numpunct(size_t refs = 0);

    char_type    decimal_point()   const;
    char_type    thousands_sep()   const;
    string       grouping()        const;
    string_type  truename()        const;
    string_type  falsename()       const;

    static locale::id id;

  protected:
    ~numpunct();                // virtual
    virtual char_type    do_decimal_point() const;
    virtual char_type    do_thousands_sep() const;
    virtual string       do_grouping()      const;
    virtual string_type  do_truename()      const;      // for bool
    virtual string_type  do_falsename()     const;      // for bool
  };
}
```

1 `numpunct<>` specifies numeric punctuation. The specializations required in Table 66 (22.1.1.1.1), namely `numpunct<wchar_t>` and `numpunct<char>`, provide classic `"C"` numeric formats, i.e. they contain information equivalent to that contained in the `"C"` locale or their wide character counterparts as if obtained by a call to `widen`.

2 The syntax for number formats is as follows, where `digit` represents the radix set specified by the `fmtflags` argument value, and `thousands-sep` and `decimal-point` are the results of corresponding `numpunct<charT>` members. Integer values have the format:

```
integer   ::= [sign] units
sign      ::= plusminus
plusminus ::= '+' | '-'
units     ::= digits [thousands-sep units]
digits    ::= digit [digits]
```

and floating-point values have:

```
floatval ::= [sign] units [decimal-point [digits]] [e [sign] digits] |
             [sign]       decimal-point  digits   [e [sign] digits]
e        ::= 'e' | 'E'
```

where the number of digits between `thousands-sep`s is as specified by `do_grouping()`. For parsing, if the `digits` portion contains no thousands-separators, no grouping constraint is applied.

22.2.3.1.1 `numpunct` members [facet.numpunct.members]

```
char_type decimal_point() const;
```

1 *Returns:* `do_decimal_point()`

```
char_type thousands_sep() const;
```

2 *Returns:* `do_thousands_sep()`

```
string grouping()  const;
```

3 *Returns:* `do_grouping()`

```
string_type truename()  const;
string_type falsename() const;
```

4 *Returns:* `do_truename()` or `do_falsename()`, respectively.

22.2.3.1.2 `numpunct` virtual functions [facet.numpunct.virtuals]

```
char_type do_decimal_point() const;
```

1 *Returns:* A character for use as the decimal radix separator. The required specializations return `'.'` or `L'.'`.

```
char_type do_thousands_sep() const;
```

2 *Returns:* A character for use as the digit group separator. The required specializations return `','` or `L','`.

```
string do_grouping() const;
```

3 *Returns:* A basic_string<char> `vec` used as a vector of integer values, in which each element `vec[i]` represents the number of digits[243] in the group at position `i`, starting with position 0 as the rightmost group. If `vec.size() <= i`, the number is the same as group `(i-1)`; if `(i<0 || vec[i]<=0 || vec[i]==CHAR_MAX)`, the size of the digit group is unlimited.

4 The required specializations return the empty string, indicating no grouping.

```
string_type do_truename()  const;
string_type do_falsename() const;
```

5 *Returns:* A string representing the name of the boolean value `true` or `false`, respectively.

6 In the base class implementation these names are `"true"` and `"false"`, or `L"true"` and `L"false"`.

22.2.3.2 Class template `numpunct_byname` [locale.numpunct.byname]

```
namespace std {
  template <class charT>
  class numpunct_byname : public numpunct<charT> {
  // this class is specialized for char and wchar_t.
  public:
    typedef charT                char_type;
    typedef basic_string<charT>  string_type;
    explicit numpunct_byname(const char*, size_t refs = 0);
    explicit numpunct_byname(const string&, size_t refs = 0);
  protected:
   ~numpunct_byname();
  };
}
```

22.2.4 The collate category [category.collate]

22.2.4.1 Class template `collate` [locale.collate]

```
namespace std {
  template <class charT>
  class collate : public locale::facet {
  public:
    typedef charT              char_type;
    typedef basic_string<charT> string_type;

    explicit collate(size_t refs = 0);

    int compare(const charT* low1, const charT* high1,
                const charT* low2, const charT* high2) const;
    string_type transform(const charT* low, const charT* high) const;
    long hash(const charT* low, const charT* high) const;

    static locale::id id;

  protected:
    ~collate();
    virtual int do_compare(const charT* low1, const charT* high1,
```

243) Thus, the string `"\003"` specifies groups of 3 digits each, and `"3"` probably indicates groups of 51 (!) digits each, because 51 is the ASCII value of `"3"`.

```
                          const charT* low2, const charT* high2) const;
    virtual string_type do_transform(const charT* low, const charT* high) const;
    virtual long do_hash (const charT* low, const charT* high) const;
  };
}
```

1 The class `collate<charT>` provides features for use in the collation (comparison) and hashing of strings. A locale member function template, `operator()`, uses the collate facet to allow a locale to act directly as the predicate argument for standard algorithms (Clause 25) and containers operating on strings. The specializations required in Table 66 (22.1.1.1.1), namely `collate<char>` and `collate<wchar_t>`, apply lexicographic ordering (25.3.8).

2 Each function compares a string of characters `*p` in the range `[low,high)`.

22.2.4.1.1 `collate` members [locale.collate.members]

```
int compare(const charT* low1, const charT* high1,
            const charT* low2, const charT* high2) const;
```

1 *Returns:* `do_compare(low1, high1, low2, high2)`

```
string_type transform(const charT* low, const charT* high) const;
```

2 *Returns:* `do_transform(low, high)`

```
long hash(const charT* low, const charT* high) const;
```

3 *Returns:* `do_hash(low, high)`

22.2.4.1.2 `collate` virtual functions [locale.collate.virtuals]

```
int do_compare(const charT* low1, const charT* high1,
               const charT* low2, const charT* high2) const;
```

1 *Returns:* `1` if the first string is greater than the second, `-1` if less, zero otherwise. The specializations required in Table 66 (22.1.1.1.1), namely `collate<char>` and `collate<wchar_t>`, implement a lexicographical comparison (25.3.8).

```
string_type do_transform(const charT* low, const charT* high) const;
```

2 *Returns:* A `basic_string<charT>` value that, compared lexicographically with the result of calling `transform()` on another string, yields the same result as calling `do_compare()` on the same two strings.[244]

```
long do_hash(const charT* low, const charT* high) const;
```

3 *Returns:* An integer value equal to the result of calling `hash()` on any other string for which `do_compare()` returns 0 (equal) when passed the two strings. [*Note:* The probability that the result equals that for another string which does not compare equal should be very small, approaching `(1.0/numeric_limits<unsigned long>::max())`. — *end note*]

22.2.4.2 Class template `collate_byname` [locale.collate.byname]

244) This function is useful when one string is being compared to many other strings.

```
namespace std {
  template <class charT>
  class collate_byname : public collate<charT> {
  public:
    typedef basic_string<charT> string_type;
    explicit collate_byname(const char*, size_t refs = 0);
    explicit collate_byname(const string&, size_t refs = 0);
  protected:
    ~collate_byname();
  };
}
```

22.2.5 The time category [category.time]

1 Templates time_get<charT,InputIterator> and time_put<charT,OutputIterator> provide date and time formatting and parsing. All specifications of member functions for time_put and time_get in the subclauses of 22.2.5 only apply to the specializations required in Tables 66 and 67 (22.1.1.1.1). Their members use their ios_base&, ios_base::iostate&, and fill arguments as described in (22.2), and the ctype<> facet, to determine formatting details.

22.2.5.1 Class template time_get [locale.time.get]

```
namespace std {
  class time_base {
  public:
    enum dateorder { no_order, dmy, mdy, ymd, ydm };
  };

  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get : public locale::facet, public time_base {
  public:
    typedef charT            char_type;
    typedef InputIterator    iter_type;

    explicit time_get(size_t refs = 0);

    dateorder date_order()  const { return do_date_order(); }
    iter_type get_time(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t)  const;
    iter_type get_date(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t)  const;
    iter_type get_weekday(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_monthname(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get_year(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm* t) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm *t, char format, char modifier = 0) const;
    iter_type get(iter_type s, iter_type end, ios_base& f,
                       ios_base::iostate& err, tm *t, const char_type *fmt, const char_type *fmtend) const;

    static locale::id id;

  protected:
```

```
    ~time_get();
    virtual dateorder do_date_order()  const;
    virtual iter_type do_get_time(iter_type s, iter_type end, ios_base&,
                                  ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_date(iter_type s, iter_type end, ios_base&,
                                  ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_weekday(iter_type s, iter_type end, ios_base&,
                                     ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_monthname(iter_type s, iter_type end, ios_base&,
                                       ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get_year(iter_type s, iter_type end, ios_base&,
                                  ios_base::iostate& err, tm* t) const;
    virtual iter_type do_get(iter_type s, iter_type end, ios_base& f,
                             ios_base::iostate& err, tm *t, char format, char modifier) const;
  };
}
```

1 `time_get` is used to parse a character sequence, extracting components of a time or date into a `struct tm` record. Each `get` member parses a format as produced by a corresponding format specifier to `time_put<>::put`. If the sequence being parsed matches the correct format, the corresponding members of the `struct tm` argument are set to the values used to produce the sequence; otherwise either an error is reported or unspecified values are assigned.[245]

2 If the end iterator is reached during parsing by any of the `get()` member functions, the member sets `ios_base::eofbit` in `err`.

22.2.5.1.1 `time_get` members [locale.time.get.members]

```
dateorder date_order() const;
```

1 *Returns:* `do_date_order()`

```
iter_type get_time(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

2 *Returns:* `do_get_time(s, end, str, err, t)`

```
iter_type get_date(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

3 *Returns:* `do_get_date(s, end, str, err, t)`

```
iter_type get_weekday(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
iter_type get_monthname(iter_type s, iter_type end, ios_base& str,
                        ios_base::iostate& err, tm* t) const;
```

4 *Returns:* `do_get_weekday(s, end, str, err, t)` or `do_get_monthname(s, end, str, err, t)`

```
iter_type get_year(iter_type s, iter_type end, ios_base& str,
                   ios_base::iostate& err, tm* t) const;
```

5 *Returns:* `do_get_year(s, end, str, err, t)`

```
iter_type get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, char format, char modifier = 0) const;
```

245) In other words, user confirmation is required for reliable parsing of user-entered dates and times, but machine-generated formats can be parsed reliably. This allows parsers to be aggressive about interpreting user variations on standard formats.

6 *Returns:* `do_get(s, end, f, err, t, format, modifier)`

```
iter_type get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, const char_type *fmt, const char_type *fmtend) const;
```

7 *Requires:* `[fmt,end)` shall be a valid range.

8 *Effects:* The function starts by evaluating `err = ios_base::goodbit`. It then enters a loop, reading zero or more characters from `s` at each iteration. Unless otherwise specified below, the loop terminates when the first of the following conditions holds:

— The expression `fmt == fmtend` evaluates to true.

— The expression `err == ios_base::goodbit` evaluates to false.

— The expression `s == end` evaluates to true, in which case the function evaluates `err = ios_base::eofbit | ios_base::failbit`.

— The next element of `fmt` is equal to `'%'`, optionally followed by a modifier character, followed by a conversion specifier character, `format`, together forming a conversion specification valid for the ISO/IEC 9945 function `strptime`. If the number of elements in the range `[fmt,fmtend)` is not sufficient to unambiguously determine whether the conversion specification is complete and valid, the function evaluates `err = ios_base::failbit`. Otherwise, the function evaluates `s = do_get(s, end, f, err, t, format, modifier)`, where the value of `modifier` is `'\0'` when the optional modifier is absent from the conversion specification. If `err == ios_base::goodbit` holds after the evaluation of the expression, the function increments `fmt` to point just past the end of the conversion specification and continues looping.

— The expression `isspace(*fmt, f.getloc())` evaluates to true, in which case the function first increments `fmt` until `fmt == fmtend || !isspace(*fmt, f.getloc())` evaluates to true, then advances `s` until `s == end || !isspace(*s, f.getloc())` is true, and finally resumes looping.

— The next character read from `s` matches the element pointed to by `fmt` in a case-insensitive comparison, in which case the function evaluates `++fmt, ++s` and continues looping. Otherwise, the function evaluates `err = ios_base::failbit`.

9 [*Note:* The function uses the `ctype<charT>` facet installed in `f`'s locale to determine valid whitespace characters. It is unspecified by what means the function performs case-insensitive comparison or whether multi-character sequences are considered while doing so.

10 *Returns:* `s`

22.2.5.1.2 `time_get` virtual functions [locale.time.get.virtuals]

```
dateorder do_date_order() const;
```

1 *Returns:* An enumeration value indicating the preferred order of components for those date formats that are composed of day, month, and year.[246] Returns `no_order` if the date format specified by `'x'` contains other variable components (e.g. Julian day, week number, week day).

```
iter_type do_get_time(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

2 *Effects:* Reads characters starting at `s` until it has extracted those `struct tm` members, and remaining format characters, used by `time_put<>::put` to produce the format specified by `"%H:%M:%S"`, or until it encounters an error or end of sequence.

246) This function is intended as a convenience only, for common formats, and may return `no_order` in valid locales.

3 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid time.

```
iter_type do_get_date(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

4 *Effects:* Reads characters starting at `s` until it has extracted those `struct tm` members and remaining format characters used by `time_put<>::put` to produce one of the following formats, or until it encounters an error. The format depends on the value returned by `date_order()` as shown in Table 77.

Table 77 — `do_get_date` effects

`date_order()`	**Format**
`no_order`	`"%m%d%y"`
`dmy`	`"%d%m%y"`
`mdy`	`"%m%d%y"`
`ymd`	`"%y%m%d"`
`ydm`	`"%y%d%m"`

5 An implementation may also accept additional implementation-defined formats.

6 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid date.

```
iter_type do_get_weekday(iter_type s, iter_type end, ios_base& str,
                         ios_base::iostate& err, tm* t) const;
iter_type do_get_monthname(iter_type s, iter_type end, ios_base& str,
                           ios_base::iostate& err, tm* t) const;
```

7 *Effects:* Reads characters starting at `s` until it has extracted the (perhaps abbreviated) name of a weekday or month. If it finds an abbreviation that is followed by characters that could match a full name, it continues reading until it matches the full name or fails. It sets the appropriate `struct tm` member accordingly.

8 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid name.

```
iter_type do_get_year(iter_type s, iter_type end, ios_base& str,
                      ios_base::iostate& err, tm* t) const;
```

9 *Effects:* Reads characters starting at `s` until it has extracted an unambiguous year identifier. It is implementation-defined whether two-digit year numbers are accepted, and (if so) what century they are assumed to lie in. Sets the `t->tm_year` member accordingly.

10 *Returns:* An iterator pointing immediately beyond the last character recognized as part of a valid year identifier.

```
iter_type do_get(iter_type s, iter_type end, ios_base& f,
    ios_base::iostate& err, tm *t, char format, char modifier) const;
```

11 *Requires:* `t` shall be dereferenceable.

12 *Effects:* The function starts by evaluating `err = ios_base::goodbit`. It then reads characters starting at `s` until it encounters an error, or until it has extracted and assigned those `struct tm` members, and any remaining format characters, corresponding to a conversion directive appropriate for the ISO/IEC 9945 function `strptime`, formed by concatenating `'%'`, the `modifier` character, when non-NUL, and the `format` character. When the concatenation fails to yield a complete valid directive

the function leaves the object pointed to by `t` unchanged and evaluates `err |= ios_base::failbit`. When `s == end` evaluates to true after reading a character the function evaluates `err |= ios_base::eofbit`.

13 For complex conversion directives such as `%c`, `%x`, or `%X`, or directives that involve the optional modifiers `E` or `O`, when the function is unable to unambiguously determine some or all `struct tm` members from the input sequence `[s,end)`, it evaluates `err |= ios_base::eofbit`. In such cases the values of those `struct tm` members are unspecified and may be outside their valid range.

14 *Remark:* It is unspecified whether multiple calls to `do_get()` with the address of the same `struct tm` object will update the current contents of the object or simply overwrite its members. Portable programs must zero out the object before invoking the function.

15 *Returns:* An iterator pointing immediately beyond the last character recognized as possibly part of a valid input sequence for the given `format` and `modifier`.

22.2.5.2 Class template `time_get_byname` [locale.time.get.byname]

```
namespace std {
  template <class charT, class InputIterator = istreambuf_iterator<charT> >
  class time_get_byname : public time_get<charT, InputIterator> {
  public:
    typedef time_base::dateorder dateorder;
    typedef InputIterator        iter_type;

    explicit time_get_byname(const char*, size_t refs = 0);
    explicit time_get_byname(const string&, size_t refs = 0);
  protected:
    ~time_get_byname();
  };
}
```

22.2.5.3 Class template `time_put` [locale.time.put]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class time_put : public locale::facet {
  public:
    typedef charT            char_type;
    typedef OutputIterator   iter_type;

    explicit time_put(size_t refs = 0);

    // the following is implemented in terms of other member functions.
    iter_type put(iter_type s, ios_base& f, char_type fill, const tm* tmb,
                  const charT* pattern, const charT* pat_end) const;
    iter_type put(iter_type s, ios_base& f, char_type fill,
                  const tm* tmb, char format, char modifier = 0) const;

    static locale::id id;

  protected:
    ~time_put();
    virtual iter_type do_put(iter_type s, ios_base&, char_type, const tm* t,
                             char format, char modifier) const;
```

```
  };
}
```

22.2.5.3.1 time_put members [locale.time.put.members]

```
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              const charT* pattern, const charT* pat_end) const;
iter_type put(iter_type s, ios_base& str, char_type fill, const tm* t,
              char format, char modifier = 0) const;
```

1 *Effects:* The first form steps through the sequence from `pattern` to `pat_end`, identifying characters that are part of a format sequence. Each character that is not part of a format sequence is written to `s` immediately, and each format sequence, as it is identified, results in a call to `do_put`; thus, format elements and other characters are interleaved in the output in the order in which they appear in the pattern. Format sequences are identified by converting each character `c` to a `char` value as if by `ct.narrow(c,0)`, where `ct` is a reference to `ctype<charT>` obtained from `str.getloc()`. The first character of each sequence is equal to `'%'`, followed by an optional modifier character `mod`[247] and a format specifier character `spec` as defined for the function `strftime`. If no modifier character is present, `mod` is zero. For each valid format sequence identified, calls `do_put(s, str, fill, t, spec, mod)`.

2 The second form calls `do_put(s, str, fill, t, format, modifier)`.

3 [*Note:* The `fill` argument may be used in the implementation-defined formats, or by derivations. A space character is a reasonable default for this argument. — *end note*]

4 *Returns:* An iterator pointing immediately after the last character produced.

22.2.5.3.2 time_put virtual functions [locale.time.put.virtuals]

```
iter_type do_put(iter_type s, ios_base&, char_type fill, const tm* t,
                 char format, char modifier) const;
```

1 *Effects:* Formats the contents of the parameter `t` into characters placed on the output sequence `s`. Formatting is controlled by the parameters `format` and `modifier`, interpreted identically as the format specifiers in the string argument to the standard library function `strftime()`.[248] except that the sequence of characters produced for those specifiers that are described as depending on the C locale are instead implementation-defined.[249]

2 *Returns:* An iterator pointing immediately after the last character produced. [*Note:* The `fill` argument may be used in the implementation-defined formats, or by derivations. A space character is a reasonable default for this argument. — *end note*]

22.2.5.4 Class template time_put_byname [locale.time.put.byname]

```
namespace std {
  template <class charT, class OutputIterator = ostreambuf_iterator<charT> >
  class time_put_byname : public time_put<charT, OutputIterator>
  {
  public:
    typedef charT          char_type;
```

247) Although the C programming language defines no modifiers, most vendors do.
248) Interpretation of the `modifier` argument is implementation-defined, but should follow POSIX conventions.
249) Implementations are encouraged to refer to other standards (such as POSIX) for these definitions.

```
    typedef OutputIterator iter_type;

    explicit time_put_byname(const char*, size_t refs = 0);
    explicit time_put_byname(const string&, size_t refs = 0);
  protected:
    ~time_put_byname();
  };
}
```

22.2.6 The monetary category [category.monetary]

1 These templates handle monetary formats. A template parameter indicates whether local or international monetary formats are to be used.

2 All specifications of member functions for `money_put` and `money_get` in the subclauses of 22.2.6 only apply to the specializations required in Tables 66 and 67 (22.1.1.1.1). Their members use their `ios_base&`, `ios_base::iostate&`, and `fill` arguments as described in (22.2), and the `moneypunct<>` and `ctype<>` facets, to determine formatting details.

22.2.6.1 Class template `money_get` [locale.money.get]

```
namespace std {
  template <class charT,
    class InputIterator = istreambuf_iterator<charT> >
  class money_get : public locale::facet {
  public:
    typedef charT               char_type;
    typedef InputIterator       iter_type;
    typedef basic_string<charT> string_type;

    explicit money_get(size_t refs = 0);

    iter_type get(iter_type s, iter_type end, bool intl,
                  ios_base& f, ios_base::iostate& err,
                  long double& units) const;
    iter_type get(iter_type s, iter_type end, bool intl,
                  ios_base& f, ios_base::iostate& err,
                  string_type& digits) const;

    static locale::id id;

  protected:
    ~money_get();
    virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                             ios_base::iostate& err, long double& units) const;
    virtual iter_type do_get(iter_type, iter_type, bool, ios_base&,
                             ios_base::iostate& err, string_type& digits) const;
  };
}
```

22.2.6.1.1 `money_get` members [locale.money.get.members]

```
iter_type get(iter_type s, iter_type end, bool intl,
              ios_base& f, ios_base::iostate& err,
              long double& quant) const;
```

```
iter_type get(s, iter_type end, bool intl, ios_base&f,
              ios_base::iostate& err, string_type& quant) const;
```

1 *Returns:* `do_get(s, end, intl, f, err, quant)`

22.2.6.1.2 `money_get` virtual functions [locale.money.get.virtuals]

```
iter_type do_get(iter_type s, iter_type end, bool intl,
                 ios_base& str, ios_base::iostate& err,
                 long double& units) const;
iter_type do_get(iter_type s, iter_type end, bool intl,
                 ios_base& str, ios_base::iostate& err,
                 string_type& digits) const;
```

1 *Effects:* Reads characters from `s` to parse and construct a monetary value according to the format specified by a `moneypunct<charT,Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. If a valid sequence is recognized, does not change `err`; otherwise, sets `err` to `(err|str.failbit)`, or `(err|str.failbit|str.eofbit)` if no more characters are available, and does not change `units` or `digits`. Uses the pattern returned by `mp.neg_format()` to parse all values. The result is returned as an integral value stored in `units` or as a sequence of digits possibly preceded by a minus sign (as produced by `ct.widen(c)` where `c` is `'-'` or in the range from `'0'` through `'9'`, inclusive) stored in `digits`. [*Example:* The sequence `$1,056.23` in a common United States locale would yield, for `units`, `105623`, or, for `digits`, `"105623"`. — *end example*] If `mp.grouping()` indicates that no thousands separators are permitted, any such characters are not read, and parsing is terminated at the point where they first appear. Otherwise, thousands separators are optional; if present, they are checked for correct placement only after all format components have been read.

2 Where `space` or `none` appears in the format pattern, except at the end, optional white space (as recognized by `ct.is`) is consumed after any required space. If `(str.flags() & str.showbase)` is false, the currency symbol is optional and is consumed only if other characters are needed to complete the format; otherwise, the currency symbol is required.

3 If the first character (if any) in the string `pos` returned by `mp.positive_sign()` or the string `neg` returned by `mp.negative_sign()` is recognized in the position indicated by `sign` in the format pattern, it is consumed and any remaining characters in the string are required after all the other format components. [*Example:* If `showbase` is off, then for a `neg` value of `"()"` and a currency symbol of `"L"`, in `"(100 L)"` the `"L"` is consumed; but if `neg` is `"-"`, the `"L"` in `"-100 L"` is not consumed. — *end example*] If `pos` or `neg` is empty, the sign component is optional, and if no sign is detected, the result is given the sign that corresponds to the source of the empty string. Otherwise, the character in the indicated position must match the first character of `pos` or `neg`, and the result is given the corresponding sign. If the first character of `pos` is equal to the first character of `neg`, or if both strings are empty, the result is given a positive sign.

4 Digits in the numeric monetary component are extracted and placed in `digits`, or into a character buffer `buf1` for conversion to produce a value for `units`, in the order in which they appear, preceded by a minus sign if and only if the result is negative. The value `units` is produced as if by[250]

```
for (int i = 0; i < n; ++i)
  buf2[i] = src[find(atoms, atoms+sizeof(src), buf1[i]) - atoms];
buf2[n] = 0;
sscanf(buf2, "%Lf", &units);
```

250) The semantics here are different from `ct.narrow`.

where n is the number of characters placed in buf1, buf2 is a character buffer, and the values src and atoms are defined as if by

```
static const char src[] = "0123456789-";
charT atoms[sizeof(src)];
ct.widen(src, src + sizeof(src) - 1, atoms);
```

Returns: An iterator pointing immediately beyond the last character recognized as part of a valid monetary quantity.

22.2.6.2 Class template money_put [locale.money.put]

```
namespace std {
  template <class charT,
    class OutputIterator = ostreambuf_iterator<charT> >
  class money_put : public locale::facet {
  public:
    typedef charT               char_type;
    typedef OutputIterator      iter_type;
    typedef basic_string<charT> string_type;

    explicit money_put(size_t refs = 0);

    iter_type put(iter_type s, bool intl, ios_base& f,
                  char_type fill, long double units) const;
    iter_type put(iter_type s, bool intl, ios_base& f,
                  char_type fill, const string_type& digits) const;

    static locale::id id;

  protected:
    ~money_put();
    virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                             long double units) const;
    virtual iter_type do_put(iter_type, bool, ios_base&, char_type fill,
                             const string_type& digits) const;
  };
}
```

22.2.6.2.1 money_put members [locale.money.put.members]

```
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill,
              long double quant) const;
iter_type put(iter_type s, bool intl, ios_base& f, char_type fill,
              const string_type& quant) const;
```

Returns: do_put(s, intl, f, loc, quant)

22.2.6.2.2 money_put virtual functions [locale.money.put.virtuals]

```
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, long double units) const;
iter_type do_put(iter_type s, bool intl, ios_base& str,
                 char_type fill, const string_type& digits) const;
```

1 *Effects:* Writes characters to `s` according to the format specified by a `moneypunct<charT,Intl>` facet reference `mp` and the character mapping specified by a `ctype<charT>` facet reference `ct` obtained from the locale returned by `str.getloc()`, and `str.flags()`. The argument `units` is transformed into a sequence of wide characters as if by

```
ct.widen(buf1, buf1 + sprintf(buf1, "%.0Lf", units), buf2)
```

for character buffers `buf1` and `buf2`. If the first character in `digits` or `buf2` is equal to `ct.widen('-')`, then the pattern used for formatting is the result of `mp.neg_format()`; otherwise the pattern is the result of `mp.pos_format()`. Digit characters are written, interspersed with any thousands separators and decimal point specified by the format, in the order they appear (after the optional leading minus sign) in `digits` or `buf2`. In `digits`, only the optional leading minus sign and the immediately subsequent digit characters (as classified according to `ct`) are used; any trailing characters (including digits appearing after a non-digit character) are ignored. Calls `str.width(0)`.

2 *Remarks:* The currency symbol is generated if and only if `(str.flags() & str.showbase)` is nonzero. If the number of characters generated for the specified format is less than the value returned by `str.width()` on entry to the function, then copies of `fill` are inserted as necessary to pad to the specified width. For the value `af` equal to `(str.flags() & str.adjustfield)`, if `(af == str.internal)` is true, the fill characters are placed where `none` or `space` appears in the formatting pattern; otherwise if `(af == str.left)` is true, they are placed after the other characters; otherwise, they are placed before the other characters. [*Note:* It is possible, with some combinations of format patterns and flag values, to produce output that cannot be parsed using `num_get<>::get`. — *end note*]

3 *Returns:* An iterator pointing immediately after the last character produced.

22.2.6.3 Class template `moneypunct` [locale.moneypunct]

```
namespace std {
  class money_base {
  public:
    enum part { none, space, symbol, sign, value };
    struct pattern { char field[4]; };
  };

  template <class charT, bool International = false>
  class moneypunct : public locale::facet, public money_base {
  public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit moneypunct(size_t refs = 0);

    charT        decimal_point() const;
    charT        thousands_sep() const;
    string       grouping()      const;
    string_type  curr_symbol()   const;
    string_type  positive_sign() const;
    string_type  negative_sign() const;
    int          frac_digits()   const;
    pattern      pos_format()    const;
    pattern      neg_format()    const;

    static locale::id id;
    static const bool intl = International;
```

```
  protected:
    ~moneypunct();
    virtual charT        do_decimal_point() const;
    virtual charT        do_thousands_sep() const;
    virtual string       do_grouping()      const;
    virtual string_type  do_curr_symbol()   const;
    virtual string_type  do_positive_sign() const;
    virtual string_type  do_negative_sign() const;
    virtual int          do_frac_digits()   const;
    virtual pattern      do_pos_format()    const;
    virtual pattern      do_neg_format()    const;
  };
}
```

1 The `moneypunct<>` facet defines monetary formatting parameters used by `money_get<>` and `money_put<>`. A monetary format is a sequence of four components, specified by a `pattern` value `p`, such that the `part` value `static_cast<part>(p.field[i])` determines the `i`th component of the format[251] In the `field` member of a `pattern` object, each value `symbol`, `sign`, `value`, and either `space` or `none` appears exactly once. The value `none`, if present, is not first; the value `space`, if present, is neither first nor last.

2 Where `none` or `space` appears, white space is permitted in the format, except where `none` appears at the end, in which case no white space is permitted. The value `space` indicates that at least one space is required at that position. Where `symbol` appears, the sequence of characters returned by `curr_symbol()` is permitted, and can be required. Where `sign` appears, the first (if any) of the sequence of characters returned by `positive_sign()` or `negative_sign()` (respectively as the monetary value is non-negative or negative) is required. Any remaining characters of the sign sequence are required after all other format components. Where `value` appears, the absolute numeric monetary value is required.

3 The format of the numeric monetary value is a decimal number:

```
value ::= units [ decimal-point [ digits ]] |
  decimal-point digits
```

if `frac_digits()` returns a positive value, or

```
value ::= units
```

otherwise. The symbol `decimal-point` indicates the character returned by `decimal_point()`. The other symbols are defined as follows:

```
units ::= digits [ thousands-sep units ]
digits ::= adigit [ digits ]
```

In the syntax specification, the symbol `adigit` is any of the values `ct.widen(c)` for `c` in the range `'0'` through `'9'`, inclusive, and `ct` is a reference of type `const ctype<charT>&` obtained as described in the definitions of `money_get<>` and `money_put<>`. The symbol `thousands-sep` is the character returned by `thousands_sep()`. The space character used is the value `ct.widen(' ')`. White space characters are those characters `c` for which `ci.is(space,c)` returns `true`. The number of digits required after the decimal point (if any) is exactly the value returned by `frac_digits()`.

4 The placement of thousands-separator characters (if any) is determined by the value returned by `grouping()`, defined identically as the member `numpunct<>::do_grouping()`.

22.2.6.3.1 `moneypunct` members [locale.moneypunct.members]

251) An array of `char`, rather than an array of `part`, is specified for `pattern::field` purely for efficiency.

```
  charT        decimal_point() const;
  charT        thousands_sep() const;
  string       grouping()      const;
  string_type  curr_symbol()   const;
  string_type  positive_sign() const;
  string_type  negative_sign() const;
  int          frac_digits()   const;
  pattern      pos_format()    const;
  pattern      neg_format()    const;
```

1 Each of these functions F returns the result of calling the corresponding virtual member function do_F().

22.2.6.3.2 moneypunct virtual functions [locale.moneypunct.virtuals]

```
charT do_decimal_point() const;
```

1 *Returns:* The radix separator to use in case do_frac_digits() is greater than zero.[252]

```
charT do_thousands_sep() const;
```

2 *Returns:* The digit group separator to use in case do_grouping() specifies a digit grouping pattern.[253]

```
string do_grouping() const;
```

3 *Returns:* A pattern defined identically as, but not necessarily equal to, the result of numpunct<charT>::do_grouping().[254]

```
string_type do_curr_symbol() const;
```

4 *Returns:* A string to use as the currency identifier symbol.[255]

```
string_type do_positive_sign() const;
string_type do_negative_sign() const;
```

5 *Returns:* do_positive_sign() returns the string to use to indicate a positive monetary value;[256] do_negative_sign() returns the string to use to indicate a negative value.

```
int do_frac_digits() const;
```

6 *Returns:* The number of digits after the decimal radix separator, if any.[257]

```
pattern do_pos_format() const;
pattern do_neg_format() const;
```

7 *Returns:* The specializations required in Table 67 (22.1.1.1.1), namely moneypunct<char>, moneypunct<wchar_t>, moneypunct<char,true>, and moneypunct<wchar_t,true>, return an object of type pattern initialized to { symbol, sign, none, value }.[258]

22.2.6.4 Class template moneypunct_byname [locale.moneypunct.byname]

252) In common U.S. locales this is '.'.
253) In common U.S. locales this is ','.
254) To specify grouping by 3s, the value is "\003" *not* "3".
255) For international specializations (second template parameter true) this is typically four characters long, usually three letters and a space.
256) This is usually the empty string.
257) In common U.S. locales, this is 2.
258) Note that the international symbol returned by do_curr_sym() usually contains a space, itself; for example, "USD ".

```
namespace std {
  template <class charT, bool Intl = false>
  class moneypunct_byname : public moneypunct<charT, Intl> {
  public:
    typedef money_base::pattern pattern;
    typedef basic_string<charT> string_type;

    explicit moneypunct_byname(const char*, size_t refs = 0);
    explicit moneypunct_byname(const string&, size_t refs = 0);
  protected:
    ~moneypunct_byname();
  };
}
```

22.2.7 The message retrieval category [category.messages]

1 Class `messages<charT>` implements retrieval of strings from message catalogs.

22.2.7.1 Class template messages [locale.messages]

```
namespace std {
  class messages_base {
  public:
    typedef int catalog;
  };

  template <class charT>
  class messages : public locale::facet, public messages_base {
  public:
    typedef charT char_type;
    typedef basic_string<charT> string_type;

    explicit messages(size_t refs = 0);

    catalog open(const basic_string<char>& fn, const locale&) const;
    string_type get(catalog c, int set, int msgid,
                    const string_type& dfault) const;
    void close(catalog c) const;

    static locale::id id;

  protected:
    ~messages();
    virtual catalog do_open(const basic_string<char>&, const locale&) const;
    virtual string_type do_get(catalog, int set, int msgid,
                               const string_type& dfault) const;
    virtual void do_close(catalog) const;
  };
}
```

1 Values of type `messages_base::catalog` usable as arguments to members `get` and `close` can be obtained only by calling member `open`.

22.2.7.1.1 messages members [locale.messages.members]

```
catalog open(const basic_string<char>& name, const locale& loc) const;
```

1 *Returns:* do_open(name, loc).

```
string_type get(catalog cat, int set, int msgid,
                const string_type& dfault) const;
```

2 *Returns:* do_get(cat, set, msgid, dfault).

```
void  close(catalog cat) const;
```

3 *Effects:* Calls do_close(cat).

22.2.7.1.2 messages virtual functions [locale.messages.virtuals]

```
catalog do_open(const basic_string<char>& name,
                const locale& loc) const;
```

1 *Returns:* A value that may be passed to get() to retrieve a message, from the message catalog identified by the string name according to an implementation-defined mapping. The result can be used until it is passed to close().

2 Returns a value less than 0 if no such catalog can be opened.

3 *Remarks:* The locale argument loc is used for character set code conversion when retrieving messages, if needed.

```
string_type do_get(catalog cat, int set, int msgid,
              const string_type& dfault) const;
```

4 *Requires:* cat shall be a catalog obtained from open() and not yet closed.

5 *Returns:* A message identified by arguments set, msgid, and dfault, according to an implementation-defined mapping. If no such message can be found, returns dfault.

```
void do_close(catalog cat) const;
```

6 *Requires:* cat shall be a catalog obtained from open() and not yet closed.

7 *Effects:* Releases unspecified resources associated with cat.

8 *Remarks:* The limit on such resources, if any, is implementation-defined.

22.2.7.2 Class template messages_byname [locale.messages.byname]

```
namespace std {
  template <class charT>
  class messages_byname : public messages<charT> {
  public:
    typedef messages_base::catalog catalog;
    typedef basic_string<charT>    string_type;

    explicit messages_byname(const char*, size_t refs = 0);
    explicit messages_byname(const string&, size_t refs = 0);
  protected:
    ~messages_byname();
  };
}
```

22.2.8 Program-defined facets [facets.examples]

1 A C++ program may define facets to be added to a locale and used identically as the built-in facets. To create a new facet interface, C++ programs simply derive from `locale::facet` a class containing a static member: `static locale::id id`.

2 [*Note:* The locale member function templates verify its type and storage class. — *end note*]

3 [*Note:* This paragraph is intentionally empty. — *end note*]

4 [*Example:* Traditional global localization is still easy:

```
#include <iostream>
#include <locale>
int main(int argc, char** argv) {
  using namespace std;
  locale::global(locale(""));           // set the global locale
                                        // imbue it on all the std streams
  cin.imbue(locale());
  cout.imbue(locale());
  cerr.imbue(locale());
  wcin.imbue(locale());
  wcout.imbue(locale());
  wcerr.imbue(locale());

  return MyObject(argc, argv).doit();
}
```

— *end example*]

5 [*Example:* Greater flexibility is possible:

```
#include <iostream>
#include <locale>
int main() {
  using namespace std;
  cin.imbue(locale(""));        // the user's preferred locale
  cout.imbue(locale::classic());
  double f;
  while (cin >> f) cout << f << endl;
  return (cin.fail() != 0);
}
```

In a European locale, with input `3.456,78`, output is `3456.78`. — *end example*]

6 This can be important even for simple programs, which may need to write a data file in a fixed format, regardless of a user's preference.

7 [*Example:* Here is an example of the use of locales in a library interface.

```
// file: Date.h
#include <iosfwd>
#include <string>
#include <locale>

class Date {
public:
  Date(unsigned day, unsigned month, unsigned year);
```

```
  std::string asString(const std::locale& = std::locale());
};

std::istream& operator>>(std::istream& s, Date& d);
std::ostream& operator<<(std::ostream& s, Date d);
```

8 This example illustrates two architectural uses of class `locale`.

9 The first is as a default argument in `Date::asString()`, where the default is the global (presumably user-preferred) locale.

10 The second is in the operators `<<` and `>>`, where a locale "hitchhikes" on another object, in this case a stream, to the point where it is needed.

```
// file: Date.C
#include "Date"                 // includes <ctime>
#include <sstream>
std::string Date::asString(const std::locale& l) {
  using namespace std;
  ostringstream s; s.imbue(l);
  s << *this; return s.str();
}

std::istream& operator>>(std::istream& s, Date& d) {
  using namespace std;
  istream::sentry cerberos(s);
  if (cerberos) {
    ios_base::iostate err = goodbit;
    struct tm t;
    use_facet< time_get<char> >(s.getloc()).get_date(s, 0, s, err, &t);
    if (!err) d = Date(t.tm_day, t.tm_mon + 1, t.tm_year + 1900);
    s.setstate(err);
  }
  return s;
}
```

— *end example*]

11 A locale object may be extended with a new facet simply by constructing it with an instance of a class derived from `locale::facet`. The only member a C++ program must define is the static member `id`, which identifies your class interface as a new facet.

12 [*Example:* Classifying Japanese characters:

```
// file: <jctype>
#include <locale>
namespace My {
  using namespace std;
  class JCtype : public locale::facet {
  public:
    static locale::id id;       // required for use as a new locale facet
    bool is_kanji (wchar_t c) const;
    JCtype() { }
  protected:
    ~JCtype() { }
  };
}
```

```
// file: filt.C
#include <iostream>
#include <locale>
#include "jctype"                // above
std::locale::id My::JCtype::id; // the static JCtype member declared above.

int main() {
  using namespace std;
  typedef ctype<wchar_t> wctype;
  locale loc(locale(""),         // the user's preferred locale ...
         new My::JCtype);        // and a new feature ...
  wchar_t c = use_facet<wctype>(loc).widen('!');
  if (!use_facet<My::JCtype>(loc).is_kanji(c))
    cout << "no it isn't!" << endl;
  return 0;
}
```

13 The new facet is used exactly like the built-in facets. — *end example*]

14 [*Example:* Replacing an existing facet is even easier. Here we do not define a member `id` because we are reusing the `numpunct<charT>` facet interface:

```
// file: my_bool.C
#include <iostream>
#include <locale>
#include <string>
namespace My {
  using namespace std;
  typedef numpunct_byname<char> cnumpunct;
  class BoolNames : public cnumpunct {
  protected:
    string do_truename()  const { return "Oui Oui!"; }
    string do_falsename() const { return "Mais Non!"; }
    ~BoolNames() { }
  public:
    BoolNames(const char* name) : cnumpunct(name) { }
  };
}

int main(int argc, char** argv) {
  using namespace std;
  // make the user's preferred locale, except for...
  locale loc(locale(""), new My::BoolNames(""));
  cout.imbue(loc);
  cout << boolalpha << "Any arguments today? " << (argc > 1) << endl;
  return 0;
}
```

— *end example*]

22.3 Standard code conversion facets [locale.stdcvt]

1 The header `<codecvt>` provides code conversion facets for various character encodings.

2 **Header `<codecvt>` synopsis**

```
namespace std {
  enum codecvt_mode {
    consume_header = 4,
    generate_header = 2,
    little_endian = 1
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff,
    codecvt_mode Mode = (codecvt_mode)0>
  class codecvt_utf8
    : public codecvt<Elem, char, mbstate_t> {
    // unspecified
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff,
    codecvt_mode Mode = (codecvt_mode)0>
  class codecvt_utf16
    : public codecvt<Elem, char, mbstate_t> {
    // unspecified
  };

  template<class Elem, unsigned long Maxcode = 0x10ffff,
    codecvt_mode Mode = (codecvt_mode)0>
  class codecvt_utf8_utf16
    : public codecvt<Elem, char, mbstate_t> {
    // unspecified
  };
}
```

3 For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:

— `Elem` is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`.

— `Maxcode` is the largest wide-character code that the facet will read or write without reporting a conversion error.

— If `(Mode & consume_header)`, the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.

— If `(Mode & generate_header)`, the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.

— If `(Mode & little_endian)`, the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.

4 For the facet `codecvt_utf8`:

— The facet shall convert between UTF-8 multibyte sequences and UCS2 or UCS4 (depending on the size of `Elem`) within the program.

— Endianness shall not affect how multibyte sequences are read or written.

— The multibyte sequences may be written as either a text or a binary file.

5 For the facet `codecvt_utf16`:

— The facet shall convert between UTF-16 multibyte sequences and UCS2 or UCS4 (depending on the size of `Elem`) within the program.

— Multibyte sequences shall be read or written according to the `Mode` flag, as set out above.

— The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

6 For the facet `codecvt_utf8_utf16`:

— The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.

— Endianness shall not affect how multibyte sequences are read or written.

— The multibyte sequences may be written as either a text or a binary file.

See also: ISO/IEC 10646-1:1993.

22.4 C Library Locales [c.locales]

1 Table 78 describes header `<clocale>`.

Table 78 — Header `<clocale>` synopsis

Type	Name(s)		
Macros:	`LC_ALL`	`LC_COLLATE`	`LC_CTYPE`
	`LC_MONETARY`	`LC_NUMERIC`	`LC_TIME`
	`NULL`		
Struct:	`lconv`		
Functions:	`localeconv`	`setlocale`	

2 The contents are the same as the Standard C library header `<locale.h>`.

See also: ISO C Clause 7.4.

23 Containers library [containers]

1 This Clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe container concepts, and components for sequence containers and associative containers, as summarized in Table 79.

Table 79 — Containers library summary

Subclause		Header(s)
23.1	Requirements	
23.1.6	Concepts	<container_concepts>

23.2	Sequence containers	`<array>`
		`<deque>`
		`<list>`
		`<queue>`
		`<stack>`
		`<vector>`

23.3	Associative containers	`<map>` `<set>`
23.4	Unordered associative containers	`<unordered_map>` `<unordered_set>`

23.1 Container requirements [container.requirements]

23.1.1 General container requirements [container.requirements.general]

1 Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.

2 All of the complexity requirements in this Clause are stated solely in terms of the number of operations on the contained objects. [*Example:* the copy constructor of type `vector <vector<int> >` has linear complexity, even though the complexity of copying each contained `vector<int>` is itself linear. — *end example*]

3 Objects stored in these components shall be constructed using `construct_element` (20.7.10) and destroyed using the `destroy` member function of the container's allocator (20.7.2.2) unless otherwise specified. A container may directly call constructors and destructors for its stored objects, without calling the `construct_element` or `destroy` functions, if the allocator models the `MinimalAllocator` concep. [*Note:* If the component is instantiated with a scoped allocator of type `A` (i.e., an allocator that meets the requirements of the `ScopedAllocator` concept), then `construct_element` may pass an inner allocator argument to `T`'s constructor. — *end note*]

4 In Tables 80 and 81, `X` denotes a container class containing objects of type `T`, `a` and `b` denote values of type `X`, `u` denotes an identifier, `r` denotes an lvalue or a const rvalue of type `X`, and `rv` denotes a non-const rvalue of type `X`.

Table 80 — Container requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`X::value_type`	`T`			compile time
`X::reference`	lvalue of `T`			compile time
`X::const_reference`	const lvalue of `T`			compile time
`X::iterator`	iterator type whose value type is `T`		meets the requirements of the `ForwardIterator` concept; convertible to `X::const_iterator`.	compile time
`X::const_iterator`	constant iterator type whose value type is `T`		meets the requirements of the `ForwardIterator` concept.	compile time
`X::difference_type`	signed integral type		is identical to the difference type of `X::iterator` and `X::const_iterator`	compile time
`X::size_type`	unsigned integral type		`size_type` can represent any non-negative value of `difference_type`	compile time
`X u;`			post: `u.size() == 0`	constant
`X();`			`X().size() == 0`	constant
`X(a);`			post: `a == X(a)`.	linear
`X u(a);` `X u = a;`			post: `u == a`	linear
`X u(rv);` `X u = rv;`			post: `u` shall be equal to the value that `rv` had before this construction	(Note B)
`a = rv;`	`X&`	All existing elements of `a` are either move assigned or destroyed	`a` shall be equal to the value that `rv` had before this construction	linear
`(&a)->~X();`	`void`		note: the destructor is applied to every element of `a`; all the memory is deallocated.	linear

Table 80 — Container requirements (continued)

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition	Complexity
`a.begin();`	`iterator`; `const_iterator` for constant `a`			constant
`a.end();`	`iterator`; `const_iterator` for constant `a`			constant
`a.cbegin();`	`const_iterator`	`const_cast<X const&>(a).begin();`		constant
`a.cend();`	`const_iterator`	`const_cast<X const&>(a).end();`		constant
`a == b`	convertible to `bool`	`==` is an equivalence relation. `a.size() == b.size() && equal(a.begin(), a.end(), b.begin())`		linear
`a != b`	convertible to `bool`	Equivalent to: `!(a == b)`		linear
`a.swap(b);`	`void`		`swap(a,b)`	(Note A)
`r = a`	`X&`		post: `r == a`.	linear
`a.size()`	`size_type`	`a.end() − a.begin()`		(Note A)
`a.max_size()`	`size_type`	`size()` of the largest possible container		(Note A)
`a.empty()`	convertible to `bool`	`a.size() == 0`		constant
`a < b`	convertible to `bool`	`lexicographical_compare(a.begin(), a.end(), b.begin(), b.end())`	pre: `<` is defined for values of `T`. `<` is a total ordering relationship.	linear
`a > b`	convertible to `bool`	`b < a`		linear
`a <= b`	convertible to `bool`	`!(a > b)`		linear
`a >= b`	convertible to `bool`	`!(a < b)`		linear

Notes: the algorithms `swap()`, `equal()` and `lexicographical_compare()` are defined in Clause 25. Those entries marked "(Note A)" should have constant complexity. Those entries marked "(Note B)" have constant complexity unless `allocator_propagate_never<X::allocator_type>::value` is `true`, in which case they have linear complexity.

5 The member function `size()` returns the number of elements in the container. Its semantics is defined by the rules of constructors, inserts, and erases.

6 `begin()` returns an iterator referring to the first element in the container. `end()` returns an iterator which is the past-the-end value for the container. If the container is empty, then `begin() == end()`;

7 In the expressions

```
i == j
i != j
i < j
i <= j
i >= j
i > j
i - j
```

where `i` and `j` denote objects of a container's `iterator` type, either or both may be replaced by an object of the container's `const_iterator` type referring to the same element with no change in semantics.

8 Copy and move constructors for all container types defined in this Clause obtain an allocator by calling `allocator_propagation_map<allocator_type>::select_for_copy_construction()` on their respective first parameters. All other constructors for these container types take an `Allocator&` argument (20.7.2.2), an allocator whose value type is the same as the container's value type. A copy of this argument is used for any memory allocation performed, by these constructors and by all member functions, during the lifetime of each container object or until the allocator is replaced. The allocator may be replaced only via assignment or `swap()`. Allocator replacement is performed by calling `allocator_propagation_map<allocator_type>::move_assign()`, `allocator_propagation_map<allocator_type>::copy_assign()`, or `allocator_propagation_map<alloc- ator_type>::swap()` within the implementation of the corresponding container operation. In all container types defined in this Clause, the member `get_allocator()` returns a copy of the `Allocator` object used to construct the container, or to replace the allocator.

9 If the iterator type of a container meets the requirements of the `BidirectionalIterator` concept, the container is called *reversible* and satisfies the additional requirements in Table 81.

Table 81 — Reversible container requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::reverse_- iterator`	iterator type pointing to `T`	`reverse_iterator<iterator>`	compile time
`X::const_- reverse_- iterator`	iterator type pointing to `const T`	`reverse_iterator<const_- iterator>`	compile time
`a.rbegin()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(end())`	constant
`a.rend()`	`reverse_iterator`; `const_reverse_iterator` for constant `a`	`reverse_iterator(begin())`	constant
`a.crbegin();`	`const_reverse_iterator`	`const_cast<X const&>(a).rbegin();`	constant
`a.crend();`	`const_reverse_iterator`	`const_cast<X const&>(a).rend();`	constant

10 Unless otherwise specified (see 23.1.4.1, 23.1.5.1, 23.2.2.3, and 23.2.6.4) all container types defined in this Clause meet the following additional requirements:

- if an exception is thrown by an `insert()` function while inserting a single element, that function has no effects.
- if an exception is thrown by a `push_back()` or `push_front()` function, that function has no effects.
- no `erase()`, `pop_back()` or `pop_front()` function throws an exception.
- no copy constructor or assignment operator of a returned iterator throws an exception.
- no `swap()` function throws an exception.
- no `swap()` function invalidates any references, pointers, or iterators referring to the elements of the containers being swapped.

11 Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

12 An object bound to an rvalue reference parameter of a member function of a container shall not be an element of that container; no diagnostic required.

13 All of the containers defined in this Clause and in Clause (21.3) except `array` meet the additional requirements of an allocator-aware container, as described in Table 82.

14

In Table 82, `X` denotes an allocator-aware container class with a `value_type` of `T` using allocator of type `A`, `u` denotes a variable, `t` denotes an lvalue or a const rvalue of type X, `rv` denotes a non-const rvalue of type `X`, `m` is a value of type `A`, and `Q` is an allocator type.

Table 82 — Allocator-aware container requirements

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`allocator_type`	`A`	*Requires:* `allocator_type::value_type` is the same as `X::value_type`.	compile time
`uses_allocator<X, Q>`	derived from `true_type` or `false_type`	`true_type` if `Q` is convertible to `A`	compile time
`constructible_with_allocator_suffix<X>`	derived from `true_type`		compile time
`get_allocator()`	`A`		constant
`X()` `X u;`		*Requires:* `A` is `DefaultConstructible` post: `u.size() == 0`, `get_allocator() == A()`	constant
`X(m)` `X u(m);`		post: `u.size() == 0`, `get_allocator() == m`	constant

Table 82 — Allocator-aware container requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity

`X(t, m)`
 	
`X u(t, m);`
		Requires: `ConstructibleAsElement<A, T, T>`
		post: `u == t`, `get_allocator() == m`
			linear

`X(rv, m)`
 	
`X u(rv, m);`
		Requires: `ConstructibleAsElement <A, T, T&&>`
		post: `u` shall be equal to the value that `rv` had before this construction, `get_allocator() == m`
			constant if `m == rv.get_allocator()`, otherwise linear

23.1.2 Container data races [container.requirements.dataraces]

1 For purposes of avoiding data races (17.6.5.7), implementations shall consider the following functions to be `const`: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, and, except in associative containers, `operator[]`.

2 Notwithstanding (17.6.5.7), implementations are required to avoid data races when the contents of the contained object in different elements in the same sequence are modified concurrently.

3 [*Note:* For a `vector<int> x` with a size greater than one, `x[1] = 5` and `*x.begin() = 10` can be executed concurrently without a data race, but `x[0] = 5` and `*x.begin() = 10` executed concurrently may result in a data race. — *end note*]

23.1.3 Sequence containers [sequence.reqmts]

1 A sequence container organizes a finite set of objects, all of the same type, into a strictly linear arrangement. The library provides three basic kinds of sequence containers: `vector`, `list`, and `deque`. It also provides container adaptors that make it easy to construct abstract data types, such as `stack`s or `queue`s, out of the basic sequence container kinds (or out of other kinds of sequence containers that the user might define).

2 `vector`, `list`, and `deque` offer the programmer different complexity trade-offs and should be used accordingly. `vector` is the type of sequence container that should be used by default. `list` should be used when there are frequent insertions and deletions from the middle of the sequence. `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.

3 In Tables 83 and 84, `X` denotes a sequence container class, `a` denotes a value of `X` containing elements of type `T`, `A` denotes `X::allocator_type` if it exists and `std::allocator<T>` if it doesn't, `i` and `j` denote iterators satisfying input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i, j)` denotes a valid range, `il` designates an object of type `initializer_list<value_type>`, `n` denotes a value of `X::size_type`, `p` denotes a valid const iterator to `a`, `q` denotes a valid dereferenceable const iterator to `a`, `[q1, q2)` denotes a valid range of const iterators in `a`, `t` denotes an lvalue or a const rvalue of `X::value_type`, and `rv` denotes a non-const rvalue of `X::value_type`. `Args` denotes a template parameter pack; `args` denotes a function parameter pack with the pattern `Args&&`.

4 The complexities of the expressions are sequence dependent.

Table 83 — Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X a(n, t)`		post: `size() == n` Constructs a sequence container with `n` copies of `t`
`X(i, j)` `X a(i, j)`		*Requires:* Each iterator in the range `[i,j)` shall be dereferenced exactly once. post: `size()` == `distance` between `i` and `j` Constructs a sequence container equal to the range `[i, j)`
`X(il);`		Equivalent to `X(il.begin(), il.end())`
`a = il;`	`X&`	`a = X(il);` `return *this;`
`a.emplace(p, args);`	`iterator`	Inserts an object of type `T` constructed with `std::forward<Args>(args)...`.
`a.insert(p,t)`	`iterator`	Inserts a copy of `t` before `p`.
`a.insert(p,rv)`	`iterator`	Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`void`	Inserts `n` copies of `t` before `p`.
`a.insert(p,i,j)`	`void`	Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: `i` and `j` are not iterators into `a`. Inserts copies of elements in `[i, j)` before `p`
`a.insert(p, il);`	`void`	`a.insert(p, il.begin(), il.end())`.
`a.erase(q)`	`iterator`	Erases the element pointed to by `q`
`a.erase(q1,q2)`	`iterator`	Erases the elements in the range `[q1, q2)`.
`a.clear()`	`void`	`erase(begin(), end())` post: `size() == 0`
`a.assign(i,j)`	`void`	*Requires:* Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: `i`, `j` are not iterators into `a`. Replaces elements in `a` with a copy of `[i, j)`.
`a.assign(il)`	`void`	`a.assign(il.begin(), il.end())`.
`a.assign(n,t)`	`void`	pre: `t` is not a reference into `a`. Replaces elements in `a` with `n` copies of `t`.

5 `iterator` and `const_iterator` types for sequence containers shall meet the requirements of the `ForwardIterator` concept.

6 The iterator returned from `a.insert(p,t)` points to the copy of `t` inserted into `a`.

7 The iterator returned from `a.erase(q)` points to the element immediately following `q` prior to the element being erased. If no such element exists, `a.end()` is returned.

8 The iterator returned by `a.erase(q1,q2)` points to the element pointed to by `q2` prior to any elements being erased. If no such element exists, `a.end()` is returned.

9 For every sequence container defined in this Clause and in Clause 21:

— If the constructor

```
template <class InputIterator>
X(InputIterator first, InputIterator last,
  const allocator_type& alloc = allocator_type())
```

is called with a type `InputIterator` that does not qualify as an input iterator, then the constructor will behave as if the overloaded constructor:

```
X(size_type, const value_type& = value_type(),
  const allocator_type& = allocator_type())
```

were called instead, with the arguments `static_cast<size_type>(first)`, `last` and `alloc`, respectively.

— If the member functions of the forms:

```
template <class InputIterator>          // such as insert()
rt fx1(iterator p, InputIterator first, InputIterator last);

template <class InputIterator>          // such as append(), assign()
rt fx2(InputIterator first, InputIterator last);

template <class InputIterator>          // such as replace()
rt fx3(iterator i1, iterator i2, InputIterator first, InputIterator last);
```

are called with a type `InputIterator` that does not qualify as an input iterator, then these functions will behave as if the overloaded member functions:

```
rt fx1(iterator, size_type, const value_type&);

rt fx2(size_type, const value_type&);

rt fx3(iterator, iterator, size_type, const value_type&);
```

were called instead, with the same arguments.

10 In the previous paragraph the alternative binding will fail if `first` is not implicitly convertible to `X::size_type` or if `last` is not implicitly convertible to `X::value_type`.

11 The extent to which an implementation determines that a type cannot be an input iterator is unspecified, except that as a minimum integral types shall not qualify as input iterators.

12 Table 84 lists operations that are provided for some types of sequence containers but not others. An implementation shall provide these operations for all container types shown in the "container" column, and shall implement them so as to take amortized constant time.

Table 84 — Optional sequence container operations

Expression	Return type	Operational semantics	Container
`a.front()`	`reference; const_reference` for constant `a`	`*a.begin()`	`vector`, `list`, `deque`, `basic_string`

Table 84 — Optional sequence container operations (continued)

Expression	Return type	Operational semantics	Container
a.back()	reference; const_reference for constant a	{ iterator tmp = a.end(); --tmp; return *tmp; }	vector, list, deque, basic_string
a.emplace_front(args)	void	a.emplace(a.begin(), std::forward<Args>(args)...)	list, deque
a.emplace_back(args)	void	a.emplace(a.end(), std::forward<Args>(args)...)	list, deque, vector
a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
a.push_front(rv)	void	a.insert(a.begin(), t)	list, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque, basic_string
a.push_back(rv)	void	a.insert(a.end(), t)	vector, list, deque, basic_string
a.pop_front()	void	a.erase(a.begin())	list, deque
a.pop_back()	void	{ iterator tmp = a.end(); --tmp; a.erase(tmp); }	vector, list, deque, basic_string
a[n]	reference; const_reference for constant a	*(a.begin() + n)	vector, deque, basic_string, match_results
a.at(n)	reference; const_reference for constant a	*(a.begin() + n)	vector, deque

13 The member function at() provides bounds-checked access to container elements. at() throws out_of_range if n >= a.size().

23.1.4 Associative containers [associative.reqmts]

1 Associative containers provide fast retrieval of data based on keys. The library provides four basic kinds of associative containers: set, multiset, map and multimap.

2 Each associative container is parameterized on Key and an ordering relation Compare that induces a strict weak ordering (25.3) on elements of Key. In addition, map and multimap associate an arbitrary type T with the Key. The object of type Compare is called the *comparison object* of a container. This comparison object may be a pointer to function or an object of a type with an appropriate function call operator. If the Compare type uses an allocator, then it conforms to the same rules as a container item; the container will construct the comparison object with the allocator appropriate to the allocator-related traits of the Compare type and whether is_scoped_allocator is true for the container's allocator type.

3 The phrase "equivalence of keys" means the equivalence relation imposed by the comparison and *not* the `operator==` on keys. That is, two keys `k1` and `k2` are considered to be equivalent if for the comparison object `comp`, `comp(k1, k2) == false && comp(k2, k1) == false`. For any two keys `k1` and `k2` in the same container, calling `comp(k1, k2)` shall always return the same value.

4 An associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. The `set` and `map` classes support unique keys; the `multiset` and `multimap` classes support equivalent keys. For `multiset` and `multimap`, `insert` and `erase` preserve the relative ordering of equivalent elements.

5 For `set` and `multiset` the value type is the same as the key type. For `map` and `multimap` it is equal to `pair<const Key, T>`. Keys in an associative container are immutable.

6 `iterator` of an associative container meets the requirements of the `BidirectionalIterator` concept. For associative containers where the value type is the same as the key type, both `iterator` and `const_iterator` are constant iterators. It is unspecified whether or not `iterator` and `const_iterator` are the same type.

7 In Table 85, `X` denotes an associative container class, `a` denotes a value of `X`, `a_uniq` denotes a value of `X` when `X` supports unique keys, `a_eq` denotes a value of `X` when `X` supports multiple keys, `u` denotes an identifier, `r` denotes an lvalue or a const rvalue of type `X`, `rv` denotes a non-const rvalue of type `X`, `i` and `j` satisfy input iterator requirements and refer to elements implicitly convertible to `value_type`, `[i,j)` denotes a valid range, `p` denotes a valid const iterator to `a`, `q` denotes a valid dereferenceable const iterator to `a`, `[q1, q2)` denotes a valid range of const iterators in `a`, `il` designates an object of type `initializer_list<value_type>`, `t` denotes a value of `X::value_type`, `k` denotes a value of `X::key_type` and `c` denotes a value of type `X::key_compare`. `A` denotes the storage allocator used by `X`, if any, or `std::allocator<X::value_type>` otherwise, and `m` denotes an allocator of a type convertible to `A`.

Table 85 — Associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`		compile time
`X::key_compare`	`Compare`	defaults to `less<key_type>`	compile time
`X::value_compare`	a binary predicate type	is the same as `key_compare` for `set` and `multiset`; is an ordering relation on pairs induced by the first component (*i.e.* `Key`) for `map` and `multimap`.	compile time
`X(c)` `X a(c);`		Constructs an empty container. Uses a copy of `c` as a comparison object.	constant
`X()` `X a;`		Constructs an empty container. Uses `Compare()` as a comparison object	constant

Table 85 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i,j,c)` `X a(i,j,c);`		Constructs an empty container and inserts elements from the range `[i, j)` into it; uses `c` as a comparison object.	$N \log N$ in general (N is the distance from `i` to `j`); linear if `[i, j)` is sorted with `value_comp()`
`X(i,j)` `X a(i,j);`		Same as above, but uses `Compare()` as a comparison object	same as above
`X(il);`		Same as `X(il.begin(), il.end())`.	same as `X(il.begin(), il.end())`.
`a = il`	`X&`	`a = X(il);` `return *this;`	constant
`a.key_comp()`	`X::key_compare`	returns the comparison object out of which `a` was constructed.	constant
`a.value_comp()`	`X::value_compare`	returns an object of `value_compare` constructed out of the comparison object	constant
`a_uniq.emplace(args)`	`pair<iterator, bool>`	inserts a `T` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	logarithmic
`a_eq.emplace(args)`	`iterator`	inserts a `T` object `t` constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element.	logarithmic

Table 85 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.emplace_hint(p, args)`	`iterator`	equivalent to `a.emplace( std::forward<Args>(args)...)` Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The `const_iterator p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	logarithmic in general, but amortized constant if the element is inserted right after `r`
`a_uniq.insert(t)`	`pair<iterator, bool>`	inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the `iterator` component of the pair points to the element with key equivalent to the key of `t`.	logarithmic
`a_eq.insert(t)`	`iterator`	inserts `t` and returns the iterator pointing to the newly inserted element. If a range containing elements equivalent to `t` exists in `a_eq`, `t` is inserted at the end of that range.	logarithmic
`a.insert(p, t)`	`iterator`	inserts `t` if and only if there is no element with key equivalent to the key of `t` in containers with unique keys; always inserts `t` in containers with equivalent keys. always returns the iterator pointing to the element with key equivalent to the key of `t`. `t` is inserted as close as possible to the position just prior to `p`.	logarithmic in general, but amortized constant if `t` is inserted right before `p`.

Table 85 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.insert(i, j)`	`void`	pre: `i`, `j` are not iterators into `a`. inserts each element from the range `[i,j)` if and only if there is no element with key equivalent to the key of that element in containers with unique keys; always inserts that element in containers with equivalent keys.	$N \log(\text{size}() + N)$ (N is the distance from `i` to `j`)
`a.insert(il)`	`void`	Equivalent to `a.insert(il.begin(), il.end())`.	
`a.erase(k)`	`size_type`	erases all elements in the container with key equivalent to `k`. returns the number of erased elements.	$\log(\text{size}()) + \text{count}(k)$
`a.erase(q)`	`iterator`	erases the element pointed to by `q`. Returns an iterator pointing to the element immediately following `q` prior to the element being erased. If no such element exists, returns `a.end()`.	amortized constant
`a.erase(q1, q2)`	`iterator`	erases all the elements in the range `[q1,q2)`. Returns `q2`.	$\log(\text{size}()) + N$ where N is the distance from `q1` to `q2`.
`a.clear()`	`void`	`erase(a.begin(),a.end())` post: `size() == 0`	linear in `size()`.
`a.find(k)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to an element with the key equivalent to `k`, or `a.end()` if such an element is not found	logarithmic
`a.count(k)`	`size_type`	returns the number of elements with key equivalent to `k`	$\log(\text{size}()) + \text{count}(k)$
`a.lower_bound(k)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to the first element with key not less than `k`, or `a.end()` if such an element is not found.	logarithmic
`a.upper_bound(k)`	`iterator`; `const_iterator` for constant `a`.	returns an iterator pointing to the first element with key greater than `k`, or `a.end()` if such an element is not found.	logarithmic

Table 85 — Associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for constant `a`.	equivalent to `make_pair(a.lower_bound(k), a.upper_bound(k))`.	logarithmic

8 The insert members shall not affect the validity of iterators and references to the container, and the erase members shall invalidate only iterators and references to the erased elements.

9 The fundamental property of iterators of associative containers is that they iterate through the containers in the non-descending order of keys where non-descending is defined by the comparison that was used to construct them. For any two dereferenceable iterators `i` and `j` such that distance from `i` to `j` is positive,

```
value_comp(*j, *i) == false
```

10 For associative containers with unique keys the stronger condition holds,

```
value_comp(*i, *j) != false.
```

11 When an associative container is constructed by passing a comparison object the container shall not store a pointer or reference to the passed object, even if that object is passed by reference. When an associative container is copied, either through a copy constructor or an assignment operator, the target container shall then use the comparison object from the container being copied, as if that comparison object had been passed to the target container in its constructor.

23.1.4.1 Exception safety guarantees [associative.reqmts.except]

1 For associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Pred` object (if any).

2 For associative containers, if an exception is thrown by any operation from within an `insert()` function inserting a single element, the `insert()` function has no effect.

3 For associative containers, no `swap` function throws an exception unless that exception is thrown by the copy constructor or copy assignment operator of the container's `Pred` object (if any).

23.1.5 Unordered associative containers [unord.req]

1 Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four unordered associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

2 Unordered associative containers conform to the requirements for Containers (23.1), except that the expressions in table 86 are not required to be valid, where a and b denote values of a type X, and X is an unordered associative container class:

Table 86 — Container requirements that are not required for unordered associative containers

Unsupported expressions
`a == b`
`a != b`
`a < b`
`a > b`
`a <= b`
`a >= b`

3 Each unordered associative container is parameterized by `Key`, by a function object `Hash` that acts as a hash function for values of type `Key`, and by a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* `T` with the `Key`. If the `Hash` or the `Pred` type uses an allocator, it shall conform to the same rules as container items; the container will construct the `Hash` and `Pred` objects with the allocator appropriate to the the allocator-related traits of the `Hash` and `Pred` types and whether `is_scoped_allocator` is true for the container's allocator type.

4 A hash function is a function object that takes a single argument of type `Key` and returns a value of type `std::size_t`.

5 Two values `k1` and `k2` of type `Key` are considered equal if the container's equality function object returns `true` when passed those values. If `k1` and `k2` are equal, the hash function shall return the same value for both.

6 An unordered associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. `unordered_set` and `unordered_map` support unique keys. `unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other. For `unordered_multiset` and `unordered_multimap`, `insert` and `erase` preserve the relative ordering of equivalent elements.

7 For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is `std::pair<const Key, T>`.

8 The elements of an unordered associative container are organized into *buckets*. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to an unordered associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements. For `unordered_multiset` and `unordered_multimap`, rehashing preserves the relative ordering of equivalent elements.

9 In table 87: `X` is an unordered associative container class, `a` is an object of type `X`, `b` is a possibly const object of type `X`, `a_uniq` is an object of type `X` when `X` supports unique keys, `a_eq` is an object of type `X` when `X` supports equivalent keys, `i` and `j` are input iterators that refer to `value_type`, `[i, j)` is a valid range, `p` and `q2` are valid const iterators to `a`, `q` and `q1` are valid dereferenceable const iterators to `a`, `[q1, q2)` is a valid range in `a`, `t` is a value of type `X::value_type`, `k` is a value of type `key_type`, `hf` is a possibly const value of type `hasher`, `eq` is a possibly const value of type `key_equal`, `n` is a value of type `size_type`, and `z` is a value of type `float`.

Table 87 — Unordered associative container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`		compile time
`X::hasher`	`Hash`		compile time
`X::key_equal`	`Pred`	`Pred` is an equivalence relation.	compile time
`X::local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::iterator`'s.	A `local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X::const_local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::const_iterator`'s.	A `const_local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X(n, hf, eq)` `X a(n, hf, eq)`	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate.	$\mathcal{O}(n)$
`X(n, hf)` `X a(n, hf)`	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate.	$\mathcal{O}(n)$
`X(n)` `X a(n)`	X	Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	$\mathcal{O}(n)$
`X()` `X a`	X	Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate.	constant
`X(i, j, n, hf, eq)` `X a(i, j, n, hf, eq)`	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `eq` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(i, j, n, hf)` `X a(i, j, n, hf)`	X	Constructs an empty container with at least `n` buckets, using `hf` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$

Table 87 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`X(i, j, n)` `X a(i, j, n)`	X	Constructs an empty container with at least `n` buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(i, j)` `X a(i, j)`	X	Constructs an empty container with an unspecified number of buckets, using `hasher()` as the hash function and `key_equal()` as the key equality predicate, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$
`X(b)` `X a(b)`	`X`	Copy constructor. In addition to the contained elements, copies the hash function, predicate, and maximum load factor.	Average case linear in `b.size()`, worst case quadratic.
`a = b`	`X`	Copy assignment operator. In addition to the contained elements, copies the hash function, predicate, and maximum load factor.	Average case linear in `b.size()`, worst case quadratic.
`b.hash_function()`	`hasher`	Returns `b`'s hash function.	constant
`b.key_eq()`	`key_equal`	Returns `b`'s key equality predicate.	constant
`a_uniq.` `emplace(args)`	`pair<iterator, bool>`	inserts a `T` object `t` constructed with `std::forward<Args>(args)...` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of `t`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq.size()`).
`a_eq. emplace(args)`	`iterator`	inserts a `T` object `t` constructed with `std::forward<Args>(args)...` and returns the iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq.size()`).

Table 87 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.emplace_hint(p, args)`	`iterator`	equivalent to `a.emplace( std::forward<Args>(args)...)`. Return value is an iterator pointing to the element with the key equivalent to the newly inserted element. The `const_iterator p` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a_uniq.insert(t)`	`pair<iterator, bool>`	Inserts `t` if and only if there is no element in the container with key equivalent to the key of `t`. The `bool` component of the returned pair indicates whether the insertion takes place, and the `iterator` component points to the element with key equivalent to the key of `t`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_uniq.size()`).
`a_eq.insert(t)`	`iterator`	Inserts `t`, and returns an iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a_eq.size()`).
`a.insert(q, t)`	`iterator`	Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of `t`. The iterator `q` is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`a.insert(i, j)`	`void`	Pre: `i` and `j` are not iterators in `a`. Equivalent to `a.insert(t)` for each element in `[i,j)`.	Average case $\mathcal{O}(N)$, where N is `distance(i, j)`. Worst case $\mathcal{O}(N *$ `a.size()`).
`a.erase(k)`	`size_type`	Erases all elements with key equivalent to `k`. Returns the number of elements erased.	Average case $\mathcal{O}$(`a.count(k)`). Worst case $\mathcal{O}$(`a.size()`).
`a.erase(q)`	`iterator`	Erases the element pointed to by `q`. Return value is the iterator immediately following `q` prior to the erasure.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).

Table 87 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`a.erase(q1, q2)`	`iterator`	Erases all elements in the range `[q1, q2)`. Return value is the iterator immediately following the erased elements prior to the erasure.	Average case linear in `distance(q1, q2)`, worst case $\mathcal{O}$(`a.size()`).
`a.clear()`	`void`	Erases all elements in the container. Post: `a.size() == 0`	Linear.
`b.find(k)`	`iterator`; `const_iterator` for const `b`.	Returns an iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`b.size()`).
`b.count(k)`	`size_type`	Returns the number of elements with key equivalent to `k`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`b.size()`).
`b.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `b`.	Returns a range containing all elements with keys equivalent to `k`. Returns `make_pair(b.end(), b.end())` if no such elements exist.	Average case $\mathcal{O}$(`b.count(k)`). Worst case $\mathcal{O}$(`b.size()`).
`b.bucket_count()`	`size_type`	Returns the number of buckets that `b` contains.	Constant
`b.max_bucket_count()`	`size_type`	Returns an upper bound on the number of buckets that `b` might ever contain.	Constant
`b.bucket(k)`	`size_type`	Returns the index of the bucket in which elements with keys equivalent to `k` would be found, if any such element existed. Post: the return value shall be in the range `[0, b.bucket_count())`.	Constant
`b.bucket_size(n)`	`size_type`	Pre: `n` shall be in the range `[0, b.bucket_count())`. Returns the number of elements in the `n` th bucket.	$\mathcal{O}$(`b.bucket_size(n)`)
`b.begin(n)`	`local_iterator`; `const_local_iterator` for const `b`.	Pre: `n` shall be in the range `[0, b.bucket_count())`. Note: `[b.begin(n), b.end(n))` is a valid range containing all of the elements in the `n` th bucket.	Constant

Table 87 — Unordered associative container requirements (in addition to container) (continued)

Expression	Return type	Assertion/note pre-/post-condition	Complexity
`b.end(n)`	`local_iterator`; `const_local_iterator` for const `b`.	Pre: `n` shall be in the range `[0, b.bucket_count())`.	Constant
`b.cbegin(n)`	`const_local_iterator`	Pre: `n` shall be in the range `[0, b.bucket_count())`. Note: `[b.cbegin(n), b.cend(n))` is a valid range containing all of the elements in the `n` th bucket.	Constant
`b.cend(n)`	`const_local_iterator`	Pre: `n` shall be in the range `[0, b.bucket_count())`.	Constant
`b.load_factor()`	`float`	Returns the average number of elements per bucket.	Constant
`b.max_load_factor()`	`float`	Returns a positive number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number.	Constant
`a.max_load_factor(z)`	`void`	Pre: `z` shall be positive. Changes the container's maximum load load factor, using `z` as a hint.	Constant
`a.rehash(n)`	`void`	Post: `a.bucket_count() > a.size() / a.max_load_factor()` and `a.bucket_count() >= n`.	Average case linear in `a.size()`, worst case quadratic.

10 Unordered associative containers are not required to support the expressions `a == b` or `a != b`. [*Note:* This is because the container requirements define operator equality in terms of equality of ranges. Since the elements of an unordered associative container appear in an arbitrary order, range equality is not a useful operation. — *end note*]

11 The iterator types `iterator` and `const_iterator` of an unordered associative container meet the requirements of the `ForwardIterator` concept. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are const iterators.

12 The insert members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The erase members shall invalidate only iterators and references to the erased elements.

13 The insert members shall not affect the validity of iterators if `(N+n) < z * B`, where `N` is the number of elements in the container prior to the insert operation, `n` is the number of elements inserted, `B` is the

container's bucket count, and `z` is the container's maximum load factor.

23.1.5.1 Exception safety guarantees [unord.req.except]

1 For unordered associative containers, no `clear()` function throws an exception. `erase(k)` does not throw an exception unless that exception is thrown by the container's `Hash` or `Pred` object (if any).

2 For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an `insert()` function inserting a single element, the `insert()` function has no effect.

3 For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the copy constructor or copy assignment operator of the container's Hash or Pred object (if any).

4 For unordered associative containers, if an exception is thrown from within a `rehash()` function other than by the container's hash function or comparison function, the `rehash()` function has no effect.

23.1.6 Container concepts [container.concepts]

1 The `container_concepts` header describes requirements on the template arguments used in container adaptors. It contains two sets of container concepts, one that uses non-member functions (23.1.6.1) and the other that uses member functions (23.1.6.2). A set of concept map templates (23.1.6.3) adapts the member-function syntax (the way most containers are implemented) to free-function syntax (which is used by most generic functions, because of its flexibility).

Header `<container_concepts>` synopsis

```
namespace std {
  // 23.1.6.1, container concepts
  concept Container<typename C> see below
  concept FrontInsertionContainer<typename C> see below
  concept BackInsertionContainer<typename C> see below
  concept StackLikeContainer<typename C> see below
  concept QueueLikeContainer<typename C> see below
  concept InsertionContainer<typename C> see below
  concept RangeInsertionContainer<typename C, typename Iter> see below
  concept FrontEmplacementContainer<typename C, typename... Args> see below
  concept BackEmplacementContainer<typename C, typename... Args> see below
  concept EmplacementContainer<typename C, typename... Args> see below

  // 23.1.6.2, member container concepts
  auto concept MemberContainer<typename C> see below
  auto concept MemberFrontInsertionContainer<typename C> see below
  auto concept MemberBackInsertionContainer<typename C> see below
  auto concept MemberStackLikeContainer<typename C> see below
  auto concept MemberQueueLikeContainer<typename C> see below
  auto concept MemberInsertionContainer<typename C> see below
  auto concept MemberRangeInsertionContainer<typename C, typename Iter> see below
  auto concept MemberFrontEmplacementContainer<typename C, typename... Args> see below
  auto concept MemberBackEmplacementContainer<typename C, typename... Args> see below
  auto concept MemberEmplacementContainer<typename C, typename... Args> see below

  // 23.1.6.3, container concept maps
  template <MemberContainer C>              concept_map Container<C> see below
  template <MemberFrontInsertionContainer C> concept_map FrontInsertionContainer<C> see below
  template <MemberBackInsertionContainer C>  concept_map BackInsertionContainer<C> see below
  template <MemberStackLikeContainer C>  concept_map StackLikeContainer<C> see below
```

```
  template <MemberQueueLikeContainer C>  concept_map QueueLikeContainer<C> see below
  template <MemberInsertionContainer C>      concept_map InsertionContainer<C> see below
  template <MemberRangeInsertionContainer C, InputIterator Iter>
    concept_map RangeInsertionContainer<C, Iter> see below
  template <MemberFrontEmplacementContainer C, typename... Args>
    concept_map FrontEmplacementContainer<C, Args...> see below
  template <MemberBackEmplacementContainer C, typename... Args>
    concept_map BackEmplacementContainer<C, Args...> see below
  template <MemberEmplacementContainer C, typename... Args>
    concept_map EmplacementContainer<C, Args...> see below
  template <typename E, size_t N>           concept_map Container<E[N]> see below
  template <typename E, size_t N>           concept_map Container<const E[N]> see below

  template<Container C> concept_map Range<C> see below;
  template<Container C> concept_map Range<const C> see below;
}
```

23.1.6.1 Free function container concepts [container.concepts.free]

1 This section contains the container concepts that are used by other parts of the library. These concepts are written in terms of free functions. For backward compatibility, member function versions and concept maps adapting member to free syntax follow in (23.1.6.2) and (23.1.6.3).

```
concept Container<typename C> {
  ObjectType           value_type      = typename C::value_type;
  typename             reference       = typename C::reference;
  typename             const_reference = typename C::const_reference;
  UnsignedIntegralLike size_type       = typename C::size_type;

  ForwardIterator iterator;
  ForwardIterator const_iterator;

  requires Convertible<reference, const_reference>
        && Convertible<reference, const value_type&>
        && Convertible<const_reference, const value_type&>;
        && Convertible<iterator, const_iterator>
        && SameType<ForwardIterator<iterator>::value_type, value_type>
        && SameType<ForwardIterator<const_iterator>::value_type, value_type>
        && Convertible<ForwardIterator<iterator>::reference, reference>
        && Convertible<ForwardIterator<const_iterator>::reference, const_reference>
        && SameType<ForwardIterator<iterator>::difference_type,
                    ForwardIterator<const_iterator>::difference_type>
        && IntegralType<size_type>
        && Convertible<ForwardIterator<iterator>::difference_type, size_type>;

  bool            empty(const C& c) { return begin(c) == end(c); }
  size_type       size(const C& c)  { return distance(begin(c), end(c)); }

  iterator        begin(C&);
  const_iterator  begin(const C&);
  iterator        end(C&);
  const_iterator  end(const C&);
  const_iterator  cbegin(const C& c) { return begin(c); }
  const_iterator  cend(const C& c)   { return end(c); }
  reference       front(C& c) { return *begin(c); }
```

```
  const_reference front(const C& c) { return *begin(c); }

  axiom AccessFront(C c) {
    if (begin(c) != end(c)) front(c) == *begin(c);
  }

  axiom ContainerSize(C c) {
    (begin(c) == end(c)) == empty(c);
    (begin(c) != end(c)) == (size(c) > 0);
  }
}
```

2 *Note:* describes a container which provides iteration through a sequence of elements stored in the container.

3 *Requires:* for a (possibly `const`-qualified) container `c`, `[begin(c), end(c))` is a valid range.

```
concept FrontInsertionContainer<typename C> : Container<C> {
  void push_front(C&, value_type&&);

  axiom FrontInsertion(C c, value_type x) {
    x == (push_front(c, x), front(c));
  }
}
```

4 *Note:* describes a container that can be modified by adding elements to the front of the sequence.

```
concept BackInsertionContainer<typename C> : Container<C> {
  void push_back(C&, value_type&&);
}
```

5 *Note:* describes a container that can be modified by adding to the back of the sequence.

```
concept StackLikeContainer<typename C> : BackInsertionContainer<C> {
  reference       back(C&);
  const_reference back(const C&);

  void pop_back(C&);

  requires BidirectionalIterator<iterator> axiom AccessBack(C c) {
    if (begin(c) != end(c)) back(c) == *(--end(c));
  }

  axiom BackInsertion(C c, value_type x) {
    x == (push_back(c, x), back(c));
  }

  axiom BackRemoval(C c, value_type x) {
    c == (push_back(c, x), pop_back(c), c);
  }
}
```

6 *Note:* describes a container that can be modified by adding or removing elements from the back of the sequence.

```
concept QueueLikeContainer<typename C> : BackInsertionContainer<C> {
  void pop_front(C&);
}
```

7 *Note:* describes a container that can be modified by adding elements to the back or removing elements from the front of the sequence.

```
concept InsertionContainer<typename C> : Container<C> {
  iterator insert(C&, const_iterator, value_type&&);

  axiom Insertion(C c, const_iterator position, value_type v) {
    v == *insert(c, position, v);
  }
}
```

8 *Note:* describes a container that can be modified by inserting elements at any position within the sequence.

```
concept RangeInsertionContainer<typename C, typename Iter> : InsertionContainer<C> {
  requires InputIterator<Iter>;
  void insert(C&, const_iterator position, Iter first, Iter last);
}
```

9 *Note:* describes a container that can be modified by inserting a sequence of elements at any position within the sequence.

```
concept FrontEmplacementContainer<typename C, typename... Args> : Container<C> {
  void emplace_front(C& c, Args&&... args);

  requires Constructible<value_type, Args...>
    axiom FrontEmplacement(C c, Args... args) {
      value_type(args...) == (emplace_front(c, args...), front(c));
    }

  requires FrontInsertionContainer<C> && Constructible<value_type, Args...>
    axiom FrontEmplacementPushEquivalence(C c, Args... args) {
      (emplace_front(c, args...), front(c)) == (push_front(c, value_type(args...)), front(c));
    }
}
```

10 *Note:* describes a container that can be modified by constructing elements at the front of the sequence.

```
concept BackEmplacementContainer<typename C, typename... Args> : Container<C> {
  void emplace_back(C& c, Args&&... args);

  requires StackLikeContainer<C> && Constructible<value_type, Args...>
    axiom BackEmplacement(C c, Args... args) {
      value_type(args...) == (emplace_back(c, args...), back(c));
    }

  requires StackLikeContainer<C> && Constructible<value_type, Args...>
    axiom BackEmplacementPushEquivalence(C c, Args... args) {
      (emplace_back(c, args...), back(c)) == (push_back(c, value_type(args...)), back(c));
    }
}
```

11 *Note:* describes a container that can be modified by constructing elements at the back of the sequence.

```
concept EmplacementContainer<typename C, typename... Args> : Container<C> {
  iterator emplace(C& c, const_iterator position, Args&&... args);
```

```
  requires Constructible<value_type, Args...>
    axiom Emplacement(C c, const_iterator position, Args... args) {
      value_type(args...) == *emplace(c, position, args...);
    }

  requires InsertionContainer<C> && Constructible<value_type, Args...>
    axiom EmplacementPushEquivalence(C c, const_iterator position, Args... args) {
      *emplace(c, position, args...) == *insert(c, position, value_type(args...));
    }
}
```

12 *Note:* describes a container that can be modified by constructing elements at any position within the sequence.

23.1.6.2 Member container concepts [container.concepts.member]

1 This section contains backward compatibility concepts, written using member function syntax, corresponding to the container concepts (23.1.6.1). Concept maps that automatically adapt these member function concepts to the free function concept syntax follow (23.1.6.3).

```
auto concept MemberContainer<typename C> {
  ObjectType           value_type      = typename C::value_type;
  typename             reference       = typename C::reference;
  typename             const_reference = typename C::const_reference;
  UnsignedIntegralLike size_type       = typename C::size_type;

  ForwardIterator iterator;
  ForwardIterator const_iterator;

  requires Convertible<reference, const_reference>
        && Convertible<reference, const value_type&>
        && Convertible<const_reference, const value_type&>;
        && Convertible<iterator, const_iterator>
        && SameType<ForwardIterator<iterator>::value_type, value_type>
        && SameType<ForwardIterator<const_iterator>::value_type, value_type>
        && Convertible<ForwardIterator<iterator>::reference, reference>
        && Convertible<ForwardIterator<const_iterator>::reference, const_reference>
        && SameType<ForwardIterator<iterator>::difference_type,
                    ForwardIterator<const_iterator>::difference_type>
        && IntegralType<size_type>
        && Convertible<ForwardIterator<iterator>::difference_type, size_type>;

  bool            C::empty() const { return  this->begin() == this->end(); }
  size_type       C::size() const  { return  distance(this->begin(), this->end()); }

  iterator        C::begin();
  const_iterator  C::begin() const;
  iterator        C::end();
  const_iterator  C::end() const;
  const_iterator  C::cbegin() const { return this->begin(); }
  const_iterator  C::cend() const   { return this->end(); }
  reference       C::front()        { return *this->begin(); }
  const_reference C::front() const  { return *this->begin(); }

  axiom MemberAccessFront(C c) {
```

```
    if (c.begin() != c.end()) c.front() == *c.begin();
  }

  axiom MemberContainerSize(C c) {
    (c.begin() == c.end()) == c.empty();
    (c.begin() != c.end()) == (c.size() > 0);
  }
}
```

2 *Note:* describes a container, in terms of member functions, which provides iteration through a sequence of elements stored in the container.

3 *Requires:* for a (possibly `const`-qualified) container `c`, `[c.begin(), c.end())` is a valid range.

```
auto concept MemberFrontInsertionContainer<typename C> : MemberContainer<C> {
  void C::push_front(value_type&&);

  axiom MemberFrontInsertion(C c, value_type x) {
    x == (c.push_front(x), c.front());
  }
}
```

4 *Note:* describes a container, in terms of member functions, that can be modified by adding elements to the front of the container.

```
auto concept MemberBackInsertionContainer<typename C> : MemberContainer<C> {
  void C::push_back(value_type&&);
}
```

5 *Note:* describes a container, in terms of member functions, that can be modified by adding elements to the back of the container.

```
auto concept MemberStackLikeContainer<typename C> : MemberBackInsertionContainer<C> {
  reference       C::back();
  const_reference C::back() const;

  void C::pop_back();

  requires BidirectionalIterator<iterator> axiom MemberAccessBack(C c) {
    if (c.begin() != c.end()) c.back() == *(--c.end());
  }

  axiom MemberBackInsertion(C c, value_type x) {
    x == (c.push_back(x), c.back());
  }

  axiom MemberBackRemoval(C c, value_type x) {
    c == (c.push_back(x), c.pop_back(), c);
  }
}
```

6 *Note:* describes a container, in terms of member functions, that can be modified by adding or removing elements from the back of the container.

```
auto concept MemberQueueLikeContainer<typename C> : MemberBackInsertionContainer<C> {
  void C::pop_front();
}
```

7 *Note:* describes a container, in terms of member functions, that can be modified by adding elements to the back or removing elements from the front of the container.

```
auto concept MemberInsertionContainer<typename C> : MemberContainer<C> {
  iterator C::insert(const_iterator, value_type&&);

  axiom MemberInsertion(C c, const_iterator position, value_type v) {
    v == *c.insert(position, v);
  }
}
```

8 *Note:* describes a container, in terms of member functions, that can be modified by inserting elements at any position within the container.

```
auto concept MemberRangeInsertionContainer<typename C, typename Iter> : MemberInsertionContainer<C> {
  requires InputIterator<Iter>;
  void C::insert(const_iterator position, Iter first, Iter last);
}
```

9 *Note:* describes a container, in terms of member functions, that can be modified by inserting a sequence of elements at any position within the sequence.

```
auto concept MemberFrontEmplacementContainer<typename C, typename... Args> : MemberContainer<C> {
  void C::emplace_front(Args&&... args);

  requires Constructible<value_type, Args...>
    axiom MemberFrontEmplacement(C c, Args... args) {
      value_type(args...) == (c.emplace_front(args...), c.front());
    }

  requires MemberFrontInsertionContainer<C> && Constructible<value_type, Args...>
    axiom MemberFrontEmplacementPushEquivalence(C c, Args... args) {
      (c.emplace_front(args...), c.front()) == (c.push_front(value_type(args...)), c.front());
    }
}
```

10 *Note:* describes a container, in terms of member functions, that can be modified by placing a newly-constructed object at the front of the sequence.

```
auto concept MemberBackEmplacementContainer<typename C, typename... Args> : MemberBackInsertionContainer<C> {
  void C::emplace_back(Args&&... args);

  requires MemberStackLikeContainer<C> && Constructible<value_type, Args...>
    axiom MemberBackEmplacement(C c, Args... args) {
      value_type(args...) == (c.emplace_back(args...), c.back());
    }

  requires MemberStackLikeContainer<C> && Constructible<value_type, Args...>
    axiom MemberBackEmplacementPushEquivalence(C c, Args... args) {
      (c.emplace_back(args...), c.back()) == (c.push_back(value_type(args...)), c.back());
    }
}
```

11 *Note:* describes a container, in terms of member functions, that can be modified by constructing elements at the back of the sequence.

```
auto concept MemberEmplacementContainer<typename C, typename... Args> : MemberInsertionContainer<C> {
```

```
  void C::emplace(const_iterator position, Args&&... args);

  requires Constructible<value_type, Args...>
    axiom MemberEmplacement(C c, const_iterator position, Args... args) {
      value_type(args...) == *c.emplace(position, args...);
    }

  requires MemberInsertionContainer<C> && Constructible<value_type, Args...>
    axiom MemberEmplacementPushEquivalence(C c, const_iterator position, Args... args) {
      *c.emplace(position, args...) == *c.insert(position, value_type(args...));
    }
}
```

12 *Note:* describes a container, in terms of member functions, that can be modified by constructing elements at any position within the sequence.

23.1.6.3 Container concept maps [container.concepts.maps]

1 This section contains concept maps that automatically adapt classes with the appropriate member functions, as specified in (23.1.6.2), to meet the free function container concept syntax in (23.1.6.1). It also contains maps adapting built-in arrays to model the appropriate container concepts, and maps adapting emplacement container concepts to to model insertion container concepts.

```
template <MemberContainer C>
concept_map Container<C> {
  typedef C::value_type      value_type;
  typedef C::reference       reference;
  typedef C::const_reference const_reference;
  typedef C::size_type       size_type;

  typedef C::iterator        iterator;
  typedef C::const_iterator  const_iterator;

  bool            empty(const C& c)   { return c.empty(); }
  size_type       size(const C& c)    { return c.size(); }

  iterator        begin(C& c)         { return c.begin(); }
  const_iterator  begin(const C& c)   { return c.begin(); }
  iterator        end(C& c)           { return c.end(); }
  const_iterator  end(const C& c)     { return c.end(); }
  const_iterator  cbegin(const C& c)  { return c.cbegin(); }
  const_iterator  cend(const C& c)    { return c.cend(); }
  reference       front(C& c)         { return c.front(); }
  const_reference front(const C& c)   { return c.front(); }
}
```

2 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `Container` concept.

```
template <MemberFrontInsertionContainer C>
concept_map FrontInsertionContainer<C> {
  typedef Container<C>::value_type value_type;

  void push_front(C& c, value_type&& v) { c.push_front(static_cast<value_type&&>(v)); }
}
```

3 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `FrontInsertionContainer` concept.

```
template <MemberBackInsertionContainer C>
concept_map BackInsertionContainer<C> {
  typedef Container<C>::value_type      value_type;

  void push_back(C& c, value_type&& v) { c.push_back(static_cast<value_type&&>(v)); }
}
```

4 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `BackInsertionContainer` concept.

```
template <MemberStackLikeContainer C>
concept_map StackLikeContainer<C> {
  typedef Container<C>::reference       reference;
  typedef Container<C>::const_reference const_reference;

  reference       back(C& c)        { return c.back(); }
  const_reference back(const C& c)  { return c.back(); }
  void            pop_back(C& c)    { c.pop_back(); }
}
```

5 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `StackLikeContainer` concept.

```
template <MemberQueueLikeContainer C>
concept_map QueueLikeContainer<C> {
    void pop_front(C& c)  { c.pop_front(); }
}
```

6 *Note:* Adapts an existing container, which uses member function syntax for each of its operations, to the `QueueLikeContainer` concept.

```
template <MemberInsertionContainer C>
concept_map InsertionContainer<C> {
  typedef Container<C>::value_type value_type;
  Container<C>::iterator insert(C& c, Container<C>::const_iterator i, value_type&& v)
  { return c.insert(i, static_cast<value_type&&>(v)); }
}
```

7 *Note:* Adapts an existing insertion container, which uses member function syntax for each of its operations, to the `InsertionContainer` concept.

```
template <MemberRangeInsertionContainer C, InputIterator Iter>
concept_map RangeInsertionContainer<C, Iter> {
  void insert(C& c, Container<C>::const_iterator i, Iter first, Iter last)
  { c.insert(i, first, last); }
}
```

8 *Note:* Adapts an existing range-insertion container, which uses member function syntax for each of its operations, to the `RangeInsertionContainer` concept.

```
template <MemberFrontEmplacementContainer C, typename... Args>
  concept_map FrontEmplacementContainer<C, Args...> {
    void emplace_front(C& c, Args&&... args)
    { c.emplace_front(forward<Args>(args)...); }
  }
```

9 *Note:* Adapts an existing front-emplace container, which uses member function syntax for each of its operations, to the `FrontEmplacementContainer` concept.

```
template <MemberBackEmplacementContainer C, typename... Args>
  concept_map BackEmplacementContainer<C, Args...> {
    void emplace_back(C& c, Args&&... args)
    { c.emplace_back(forward<Args>(args)...); }
  }
```

10 *Note:* Adapts an existing back-emplace container, which uses member function syntax for each of its operations, to the `BackEmplacementContainer` concept.

```
template <MemberEmplacementContainer C, typename... Args>
  concept_map EmplacementContainer<C, Args...> {
    Container<C>::iterator emplace(C& c, Container<C>::const_iterator position, Args&&... args)
    { return c.emplace(position, forward<Args>(args)...); }
  }
```

11 *Note:* Adapts an existing emplace container, which uses member function syntax for each of its operations, to the `EmplacementContainer` concept.

```
template <typename E, size_t N>
concept_map Container<E[N]> {
  typedef E                  value_type;
  typedef E&                 reference;
  typedef const E&           const_reference;
  typedef size_t             size_type;
  typedef E*                 iterator;
  typedef const E*           const_iterator;

  bool           empty(const E(&c)[N]) { return N==0; }
  size_type      size(const E(&c) [N]) { return N; }

  iterator       begin(E(&c)[N])       { return c; }
  const_iterator begin(const E(&c)[N]) { return c; }
  iterator       end(E(&c)[N])         { return c + N; }
  const_iterator end(const E(&c)[N])   { return c + N; }
}

template <typename E, size_t N>
concept_map Container<const E[N]> {
  typedef E                  value_type;
  typedef const E&           reference;
  typedef const E&           const_reference;
  typedef size_t             size_type;

  typedef const E*           iterator;
  typedef const E*           const_iterator;

  bool           empty(const E(&c)[N]) { return N==0; }
  size_type      size(const E(&c)[N])  { return N; }

  const_iterator begin(const E(&c)[N]) { return c; }
  const_iterator end(const E(&c)[N])   { return c + N; }
}
```

12 *Note:* Adapts built-in arrays to the `Container` concept.

```
template<Container C>
concept_map Range<C> {
  typedef C::iterator iterator;
  iterator begin(C& c) { return Container<C>::begin(c); }
  iterator end(C& c) { return Container<C>::end(c); }
}

template<Container C>
concept_map Range<const C> {
  typedef C::const_iterator iterator;
  iterator begin(const C& c) { return Container<C>::begin(c); }
  iterator end(const C& c) { return Container<C>::end(c); }
}
```

13 *Note:* these concept_maps adapt any type that meets the requirements of `Container` to the `Range` concept.

23.2 Sequence containers [sequences]

1 Headers `<array>`, `<deque>`, `<forward_list>`, `<list>`, `<queue>`, `<stack>`, and `<vector>`.

Header `<array>` synopsis

```
namespace std {
  template <ValueType T, size_t N >
    requires NothrowDestructible<T>
    struct array;
  template <EqualityComparable T, size_t N>
    bool operator==(const array<T,N>& x, const array<T,N>& y);
  template <EqualityComparable T, size_t N>
    bool operator!=(const array<T,N>& x, const array<T,N>& y);
  template <LessThanComparable T, size_t N>
    bool operator<(const array<T,N>& x, const array<T,N>& y);
  template <LessThanComparable T, size_t N>
    bool operator>(const array<T,N>& x, const array<T,N>& y);
  template <LessThanComparable T, size_t N>
    bool operator<=(const array<T,N>& x, const array<T,N>& y);
  template <LessThanComparable T, size_t N>
    bool operator>=(const array<T,N>& x, const array<T,N>& y);
  template <Swappable T, size_t N >
    void swap(array<T,N>& x, array<T,N>& y);

  template <ObjectType T> class tuple_size;
  template <size_t I, ObjectType T>
    class tuple_element;
  template <ObjectType T, size_t N>
    struct tuple_size<array<T, N> >;
  template <size_t I, class T, size_t N>
    requires True<(I < N)>
    struct tuple_element<I, array<T, N> >;
  template <size_t I, class T, size_t N>
    requires True<(I < N)>
    T& get(array<T, N>&);
  template <size_t I, class T, size_t N>
    requires True<(I < N)>
    const T& get(const array<T, N>&);
```

```
}
```

Header <deque> synopsis

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
    requires NothrowDestructible<T>
    class deque;
  template <EqualityComparable T, class Alloc>
    bool operator==(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <ObjectType T, class Alloc>
    void swap(deque<T,Alloc>& x, deque<T,Alloc>& y);
  template <ObjectType T, class Alloc>
    void swap(deque<T,Alloc>&& x, deque<T,Alloc>& y);
  template <ObjectType T, class Alloc>
    void swap(deque<T,Alloc>& x, deque<T,Alloc>&& y);
}
```

Header <forward_list> synopsis

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
    requires NothrowDestructible<T>
    class forward_list;
  template <EqualityComparable T, class Alloc>
    bool operator==(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(forward_list<T,Alloc>& x, forward_list<T,Alloc>& y);
}
```

Header <list> synopsis

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
    requires NothrowDestructible<T>
    class list;
```

```
  template <EqualityComparable T, class Alloc>
    bool operator==(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(list<T,Alloc>& x, list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(list<T,Alloc>&& x, list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(list<T,Alloc>& x, list<T,Alloc>&& y);
}
```

Header <queue> synopsis

```
namespace std {
  template <ObjectType T, class Cont = deque<T> >
    requires QueueLikeContainer<Cont>
          && SameType<T, Cont::value_type>
          && NothrowDestructible<Cont>
    class queue;
  template <class T, EqualityComparable Cont>
    bool operator==(const queue<T, Cont>& x,const queue<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator< (const queue<T, Cont>& x,const queue<T, Cont>& y);
  template <class T, EqualityComparable Cont>
    bool operator!=(const queue<T, Cont>& x,const queue<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator> (const queue<T, Cont>& x,const queue<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator>=(const queue<T, Cont>& x,const queue<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator<=(const queue<T, Cont>& x,const queue<T, Cont>& y);
  template <ObjectType T, Swappable Cont>
    void swap(queue<T,Cont>& x, queue<T,Cont>& y);
  template <ObjectType T, Swappable Cont>
    void swap(queue<T,Cont>&& x, queue<T,Cont>& y);
  template <ObjectType T, Swappable Cont>
    void swap(queue<T,Cont>& x, queue<T,Cont>&& y);

  template <ObjectType T, StackLikeContainer Cont = vector<T>,
        StrictWeakOrder<auto, T> Compare = less<typename Cont::value_type> >
    requires SameType<Cont::value_type, T> && RandomAccessIterator<Cont::iterator>
          && ShuffleIterator<Cont::iterator> && CopyConstructible<Compare>
          && NothrowDestructible<Cont>
  class priority_queue;
  template <ObjectType T, Swappable Cont, Swappable Compare>
    void swap(priority_queue<T, Cont, Compare>& x, priority_queue<T, Cont, Compare>& y);
  template <ObjectType T, Swappable Cont, Swappable Compare>
```

```
    void swap(priority_queue<T, Cont, Compare>&& x, priority_queue<T, Cont, Compare>& y);
  template <ObjectType T, Swappable Cont, Swappable Compare>
    void swap(priority_queue<T, Cont, Compare>& x, priority_queue<T, Cont, Compare>&& y);
}
```

Header <stack> synopsis

```
namespace std {
  template <ObjectType T, StackLikeContainer Cont = deque<T> >
    requires SameType<Cont::value_type, T>
          && NothrowDestructible<Cont>
    class stack;
  template <class T, EqualityComparable Cont>
    bool operator==(const stack<T, Cont>& x,const stack<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator< (const stack<T, Cont>& x,const stack<T, Cont>& y);
  template <class T, EqualityComparable Cont>
    bool operator!=(const stack<T, Cont>& x,const stack<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator> (const stack<T, Cont>& x,const stack<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator>=(const stack<T, Cont>& x,const stack<T, Cont>& y);
  template <class T, LessThanComparable Cont>
    bool operator<=(const stack<T, Cont>& x,const stack<T, Cont>& y);
  template <ObjectType T, Swappable Cont>
    void swap(stack<T,Cont>& x, stack<T, Cont>& y);
  template <ObjectType T, Swappable Cont>
    void swap(stack<T, Cont>&& x, stack<T, Cont>& y);
  template <ObjectType T, Swappable Cont>
    void swap(stack<T, Cont>& x, stack<T, Cont>&& y);
}
```

Header <vector> synopsis

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
    requires MoveConstructible<T>
    class vector;
  template <EqualityComparable T, class Alloc>
    bool operator==(const vector<T,Alloc>& x,const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const vector<T,Alloc>& x,const vector<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const vector<T,Alloc>& x,const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const vector<T,Alloc>& x,const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const vector<T,Alloc>& x,const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const vector<T,Alloc>& x,const vector<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(vector<T,Alloc>& x, vector<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(vector<T,Alloc>&& x, vector<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(vector<T,Alloc>& x, vector<T,Alloc>&& y);
```

```
template <Allocator Alloc> class vector<bool,Alloc>;
```

23.2.1 Class template array [array]

1 The header <array> defines a class template for storing fixed-size sequences of objects. An array supports random access iterators. An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant. The elements of an array are stored contiguously, meaning that if a is an array<T, N> then it obeys the identity &a[n] == &a[0] + n for all 0 <= n < N.

2 An array is an aggregate (8.5.1) that can be initialized with the syntax

```
array a<T, N> = { initializer-list };
```

where *initializer-list* is a comma separated list of up to N elements whose types are convertible to T.

3 Unless otherwise specified, all array operations are as described in 23.1. Descriptions are provided here only for operations on array that are not described in that Clause or for operations where there is additional semantic information.

```
namespace std {
  template <ValueType T, size_t N >
  requires NothrowDestrucbible<T>
  struct array {
    // types:
    typedef T &                                     reference;
    typedef const T &                               const_reference;
    typedef implementation defined                  iterator;
    typedef implementation defined                  const_iterator;
    typedef size_t                                  size_type;
    typedef ptrdiff_t                               difference_type;
    typedef T                                       value_type;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

    T       elems[N];            // exposition only

    // No explicit construct/copy/destroy for aggregate type

    requires CopyAssignable<T> void fill(const T& u);
    requeirs Swappable<T> void swap(array<T, N> &);

    // iterators:
    iterator               begin();
    const_iterator         begin() const;
    iterator               end();
    const_iterator         end() const;

    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;
    reverse_iterator       rend();
    const_reverse_iterator rend() const;

    const_iterator         cbegin() const;
    const_iterator         cend() const;
    const_reverse_iterator crbegin() const;
```

```
    const_reverse_iterator crend() const;

    // capacity:
    constexpr size_type size() const;
    constexpr size_type max_size() const;
    constexpr bool      empty() const;

    // element access:
    reference       operator[](size_type n);
    const_reference operator[](size_type n) const;
    const_reference at(size_type n) const;
    reference       at(size_type n);
    reference       front();
    const_reference front() const;
    reference       back();
    const_reference back() const;

    T *       data();
    const T * data() const;
  };
}
```

4 [*Note:* The member variable `elems` is shown for exposition only, to emphasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. — *end note*]

23.2.1.1 `array` constructors, copy, and assignment [array.cons]

1 The conditions for an aggregate (8.5.1) shall be met. Class `array` relies on the implicitly-declared special member functions (12.1, 12.4, and 12.8) to conform to the container requirements table in 23.1.

23.2.1.2 `array` specialized algorithms [array.special]

```
template <Swappable T, size_t N> void swap(array<T,N>& x, array<T,N>& y);
```

1 *Effects:*

```
swap_ranges(x.begin(), x.end(), y.begin() );
```

23.2.1.3 `array::size` [array.size]

```
constexpr size_type size();
```

1 *Returns:* `N`

23.2.1.4 `array::data` [array.data]

```
T *data();
const T *data() const;
```

1 *Returns:* `elems`.

23.2.1.5 `array::fill` [array.fill]

```
requires CopyAssignable<T> void fill(const T& u);
```

1 *Effects:* `fill_n(begin(), N, u)`

23.2.1.6 Zero sized arrays [array.zero]

1 `array` shall provide support for the special case `N == 0`.

2 In the case that `N == 0`, `begin() == end() ==` unique value. The return value of `data()` is unspecified.

3 The effect of calling `front()` or `back()` for a zero-sized array is implementation defined.

23.2.1.7 Tuple interface to class template array [array.tuple]

`tuple_size<array<T, N> >::value`

1 *Return type:* integral constant expression.

2 *Value:* `N`

`tuple_element<I, array<T, N> >::type`

3 *Requires:* `I < N`. The program is ill-formed if `I` is out of bounds.

4 *Value:* The type T.

```
template <size_t I, class T, size_t N>
  requires True<(I < N)>
  T& get(array<T, N>& a);
```

5 *Returns:* A reference to the `I`th element of `a`, where indexing is zero-based.

6 *Throws:* nothing.

```
template <size_t I, class T, size_t N>
  requires True<(I < N)>
  const T& get(const array<T, N>& a);
```

7 *Returns:* A const reference to the `I`th element of `a`, where indexing is zero-based.

8 *Throws:* nothing.

23.2.2 Class template deque [deque]

1 A `deque` is a sequence container that, like a `vector` (23.2.6), supports random access iterators. In addition, it supports constant time insert and erase operations at the beginning or the end; insert and erase in the middle take linear time. That is, a deque is especially optimized for pushing and popping elements at the beginning and end. As with vectors, storage management is handled automatically.

2 A `deque` satisfies all of the requirements of a container, of a reversible container (given in tables in 23.1), of a sequence container, including the optional sequence container requirements (23.1.3), and of an allocator-aware container (Table 82). Descriptions are provided here only for operations on `deque` that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
  requires NothrowDestructible<T>
  class deque {
  public:
    // types:
    typedef typename Alloc::reference         reference;
    typedef typename Alloc::const_reference   const_reference;
    typedef implementation-defined                iterator;       // See 23.1
    typedef implementation-defined                const_iterator; // See 23.1
```

```
typedef implementation-defined               size_type;      // See 23.1
typedef implementation-defined               difference_type;// See 23.1
typedef T                                    value_type;
typedef Alloc                               allocator_type;
typedef typename Alloc::pointer             pointer;
typedef typename Alloc::const_pointer       const_pointer;
typedef reverse_iterator<iterator>         reverse_iterator;
typedef reverse_iterator<const_iterator> const_reverse_iterator;

// 23.2.2.1 construct/copy/destroy:
explicit deque(const Alloc& = Alloc());
requires AllocatableElement<Alloc, T> explicit deque(size_type n);
requires AllocatableElement<Alloc, T, const T&>
  deque(size_type n, const T& value, const Alloc& = Alloc());
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  deque(Iter first, Iter last, const Alloc& = Alloc());
requires AllocatableElement<Alloc, T, const T&> deque(const deque<T,Alloc>& x);
requires AllocatableElement<Alloc, T, T&&> deque(deque&&);
requires AllocatableElement<Alloc, T, const T&> deque(const deque&, const Alloc&);
requires AllocatableElement<Alloc, T, T&&> deque(deque&&, const Alloc&);
requires AllocatableElement<Alloc, T, const T&>
  deque(initializer_list<T>, const Allocator& = Allocator());

~deque();
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  deque<T,Alloc>& operator=(const deque<T,Alloc>& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
  deque<T,Alloc>& operator=( deque<T,Alloc>&& x);
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  deque& operator=(initializer_list<T>);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
        && HasAssign<T, Iter::reference>
  void assign(Iter first, Iter last);
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  void assign(size_type n, const T& t);
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  void assign(initializer_list<T>);
allocator_type get_allocator() const;

// iterators:
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;
```

```
// 23.2.2.2 capacity:
size_type size() const;
size_type max_size() const;
requires AllocatableElement<Alloc, T>
  void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&>
  void resize(size_type sz, const T& c);
void shrink_to_fit();
bool empty() const;

// element access:
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
reference       at(size_type n);
const_reference at(size_type n) const;
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;

// 23.2.2.3 modifiers:
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_front(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_back(Args&&... args);

requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);

template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...> && MoveAssignable<T>
  iterator emplace(const_iterator position, Args&&... args);

requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
  iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  void insert(const_iterator position, size_type n, const T& x);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference> && MoveAssignable<T>
  void insert(const_iterator position, Iter first, Iter last);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  void insert(const_iterator position, initializer_list<T>);

void pop_front();
void pop_back();

requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

```
    void     swap(deque<T,Alloc>&&);
    void     clear();
  };

  template <EqualityComparable T, class Alloc>
    bool operator==(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const deque<T,Alloc>& x, const deque<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const deque<T,Alloc>& x, const deque<T,Alloc>& y);

  // specialized algorithms:
  template <ValueType T, class Alloc>
    void swap(deque<T,Alloc>& x, deque<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(deque<T,Alloc>&& x, deque<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(deque<T,Alloc>& x, deque<T,Alloc>&& y);
}
```

23.2.2.1 `deque` constructors, copy, and assignment [deque.cons]

```
explicit deque(const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty `deque`, using the specified allocator.

2 *Complexity:* Constant.

```
requires AllocatableElement<Alloc, T> explicit deque(size_type n);
```

3 *Effects:* Constructs a `deque` with `n` default constructed elements.

4 *Complexity:* Linear in `n`.

```
requires AllocatableElement<Alloc, T, const T&>
deque(size_type n, const T& value,
      const Alloc& = Alloc());
```

5 *Effects:* Constructs a `deque` with `n` copies of `value`, using the specified allocator.

6 *Complexity:* Linear in `n`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  deque(Iter first, Iter last,
        const Alloc& = Alloc());
```

7 *Effects:* Constructs a `deque` equal to the the range `[first,last)`, using the specified allocator.

8 *Complexity:* `distance(first, last)`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
```

```
       && HasAssign<T, Iter::reference>
  void assign(Iter first, Iter last);
```

9 *Effects:*

```
erase(begin(), end());
insert(begin(), first, last);
```

```
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
void assign(size_type n, const T& t);
```

10 *Effects:*

```
erase(begin(), end());
insert(begin(), n, t);
```

23.2.2.2 deque capacity [deque.capacity]

```
requires AllocatableElement<Alloc, T>
  void resize(size_type sz);
```

1 *Effects:* If `sz < size()`, equivalent to `erase(begin() + sz, end());`. If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

```
requires AllocatableElement<Alloc, T, const T&>
  void resize(size_type sz, const T& c);
```

2 *Effects:*

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size())
  erase(begin()+sz, end());
else
  ;                 // do nothing
```

```
void shrink_to_fit();
```

3 *Remarks:* `shrink_to_fit` is a non-binding request to reduce memory use. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*]

23.2.2.3 deque modifiers [deque.modifiers]

```
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
  iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  void     insert(const_iterator position, size_type n, const T& x);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference> && MoveAssignable<T>
  void insert(const_iterator position,
              Iter first, Iter last);

template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
```

```
  void emplace_front(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_back(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...> && MoveAssignable<T>
  iterator emplace(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);
```

1 *Effects:* An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. An insertion at either end of the deque invalidates all the iterators to the deque, but has no effect on the validity of references to elements of the deque.

2 *Remarks:* If an exception is thrown other than by the copy constructor or assignment operator of `T` there are no effects.

3 *Complexity:* The complexity is linear in the number of elements inserted plus the lesser of the distances to the beginning and end of the deque. Inserting a single element either at the beginning or end of a deque always takes constant time and causes a single call to a constructor of `T`.

```
requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

4 *Effects:* An erase in the middle of the deque invalidates all the iterators and references to elements of the deque and the past-the-end iterator. An erase at the beginning of the deque invalidates only the iterators and the references to the erased elements. An erase at the end of the deque invalides only the iterators and the references to the erased elements and the past-the-end iterator.

5 *Complexity:* The number of calls to the destructor is the same as the number of elements erased, but the number of the calls to the assignment operator is at most equal to the minimum of the number of elements before the erased elements and the number of elements after the erased elements.

6 *Throws:* Nothing unless an exception is thrown by the copy constructor or assignment operator of `T`.

23.2.2.4 `deque` specialized algorithms [deque.special]

```
template <ValueType T, class Alloc>
  void swap(deque<T,Alloc>& x, deque<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(deque<T,Alloc>&& x, deque<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(deque<T,Alloc>& x, deque<T,Alloc>&& y);
```

1 *Effects:*

```
x.swap(y);
```

23.2.3 Class template `forward_list` [forwardlist]

1 A `forward_list` is a container that supports forward iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Fast random access to list elements is not supported. [*Note:* It is intended that `forward_list` have zero space or time

overhead relative to a hand-written C-style singly linked list. Features that would conflict with that goal have been omitted. — *end note*]

2 A `forward_list` satisfies all of the requirements of a container (table 80), except that the `size()` member function is not provided. Descriptions are provided here only for operations on `forward_list` that are not described in that table or for operations where there is additional semantic information.

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
  requires NothrowDestructible<T>
  class forward_list {
  public:
    // types:
    typedef typename Alloc::reference reference;
    typedef typename Alloc::const_reference const_reference;
    typedef implementation-defined iterator;       // See 23.1
    typedef implementation-defined const_iterator; // See 23.1
    typedef implementation-defined size_type;      // See 23.1
    typedef implementation-defined difference_type;// See 23.1
    typedef T value_type;
    typedef Alloc allocator_type;
    typedef typename Alloc::pointer pointer;
    typedef typename Alloc::const_pointer const_pointer;

    // 23.2.3.1 construct/copy/destroy:
    explicit forward_list(const Alloc& = Alloc());
    requires AllocatableElement<Alloc, T>
      explicit forward_list(size_type n);
    requires AllocatableElement<Alloc, T, const T&>
      forward_list(size_type n, const T& value,
                   const Alloc& = Alloc());
    template <InputIterator Iter>
      AllocatableElement<Alloc, T, Iter::reference>
      forward_list(Iter first, Iter last,
                   const Alloc& = Alloc());
    requires AllocatableElement<Alloc, T, const T&>
      forward_list(const forward_list<T,Alloc>& x);
    requires AllocatableElement<Alloc, T, T&&> forward_list(forward_list<T,Alloc>&& x);
    requires AllocatableElement<Alloc, T, const T&>
      forward_list(initializer_list<T>, const Allocator& = Allocator());
    ~forward_list();
    requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
      forward_list<T,Alloc>& operator=(const forward_list<T,Alloc>& x);
    requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
      forward_list<T,Alloc>& operator=(forward_list<T,Alloc>&& x);
    requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
      forward_list<T,Alloc> operator=(initializer_list<T>);
    template <InputIterator Iter>
      requires AllocatableElement<Alloc, T, Iter::reference>
            && HasAssign<T, Iter::reference>
      void assign(Iter first, Iter last);
    requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
      void assign(size_type n, const T& t);
    requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
      void assign(initializer_list<T>);
    allocator_type get_allocator() const;
```

```
// 23.2.3.2 iterators:
iterator before_begin();
const_iterator before_begin() const;
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;

const_iterator cbegin() const;
const_iterator cbefore_begin() const;
const_iterator cend() const;

// capacity:
bool empty() const;
size_type max_size() const;

// 23.2.3.3 element access:
reference front();
const_reference front() const;

// 23.2.3.4 modifiers:
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_front(Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
void pop_front();

template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  iterator emplace_after(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&>
  iterator insert_after(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&>
  iterator insert_after(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&>
  void insert_after(const_iterator position, initializer_list<T> il);

requires AllocatableElement<Alloc, T, const T&>
  void insert_after(const_iterator position, size_type n, const T& x);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  void insert_after(const_iterator position, Iter first, Iter last);

iterator erase_after(const_iterator position);
iterator erase_after(const_iterator position, iterator last);
void swap(forward_list<T,Alloc>&&);

requires AllocatableElement<Alloc, T> void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, value_type c);
void clear();

// 23.2.3.5 forward_list operations:
void splice_after(const_iterator position, forward_list<T,Alloc>&& x);
```

```
    void splice_after(const_iterator position, forward_list<T,Alloc>&& x,
                      const_iterator i);
    void splice_after(const_iterator position, forward_list<T,Alloc>&& x,
                      const_iterator first, const_iterator last);

    requires EqualityComparable<T> void remove(const T& value);
    template <Predicate<auto, T> Pred> void remove_if(Pred pred);

    requires EqualityComparable<T> void unique();
    template <EquivalenceRelation<auto, T> BinaryPredicate>
      void unique(BinaryPredicate binary_pred);

    requires LessThanComparable<T> void merge(forward_list<T,Alloc>&& x);
    template <StrictWeakOrder<auto, T> Compare>
      void merge(forward_list<T,Alloc>&& x, Compare comp);

    requires LessThanComparable<T> void sort();
    template <StrictWeakOrder<auto, T> Compare> void sort(Compare comp);

    void reverse();
  };

  // Comparison operators
  template <EqualityComparable T, class Alloc>
    bool operator==(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const forward_list<T,Alloc>& x, const forward_list<T,Alloc>& y);

  // 23.2.3.6 specialized algorithms:
  template <ValueType T, class Alloc>
    void swap(forward_list<T,Alloc>& x, forward_list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(forward_list<T,Alloc>&& x, forward_list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(forward_list<T,Alloc>& x, forward_list<T,Alloc>&& y);
}
```

23.2.3.1 `forward_list` constructors, copy, assignment [forwardlist.cons]

```
explicit forward_list(const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty `forward_list` object using the specified allocator.

2 *Complexity:* Constant.

```
requires AllocatableElement<Alloc, T>
  explicit forward_list(size_type n);
```

3 *Effects:* Constructs a `forward_list` object with `n` default constructed elements.

4 *Complexity:* Linear in `n`.

```
requires AllocatableElement<Alloc, T, const T&>
  forward_list(size_type n, const T& value, const Alloc& = Alloc());
```

5 *Effects:* Constructs a `forward_list` object with `n` copies of `value` using the specified allocator.

6 *Complexity:* Linear in `n`.

```
template <InputIterator Iter>
  AllocatableElement<Alloc, T, Iter::reference>
  forward_list(Iter first, Iter last,
               const Alloc& = Alloc());
```

7 *Effects:* Constructs a `forward_list` object equal to the range `[first,last)`.

8 *Complexity:* Linear in `distance(first, last)`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
        && HasAssign<T, Iter::reference>
  void assign(Iter first, Iter last);
```

9 *Effects:* `clear(); insert_after(before_begin(), first, last);`

```
AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  void assign(size_type n, const T& t);
```

10 *Effects:* `clear(); insert_after(before_begin(), n, t);`

23.2.3.2 `forward_list` iterators [forwardlist.iter]

```
{iterator before_begin();
const_iterator before_begin() const;
const_iterator cbefore_begin() const;
```

1 *Returns:* A non-dereferenceable iterator that, when incremented, is equal to the iterator returned by `begin()`.

23.2.3.3 `forward_list` element access [forwardlist.access]

```
reference front();
const_reference front() const;
```

1 *Returns:* `*begin()`

23.2.3.4 `forward_list` modifiers [forwardlist.modifiers]

1 None of the overloads of `insert_after` shall affect the validity of iterators and reference, and `erase_after` shall invalidate only the iterators and references to the erased elements. If an exception is thrown during `insert_after` there shall be no effect. Insertion of `n` elements into a `forward_list` is linear in `n`, and the number of calls to the copy or move constructor of `T` is exactly equal to `n`. Erasing `n` elements from a `forward_list` is linear time in `n` and the number of calls to the destructor of type `T` is exactly equal to `n`.

```
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
```

```
void emplace_front(Args&&... args);
```

2 *Effects:* Inserts an object of type `value_type` constructed with `value_type(forward<Args>(args)...)` at the beginning of the list.

```
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
```

3 *Effects:* Inserts a copy of `x` at the beginning of the list.

```
void pop_front();
```

4 *Effects:* `erase_after(before_begin())`

```
requires AllocatableElement<Alloc, T, const T&>
  iterator insert_after(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&>
  iterator insert_after(const_iterator position, T&& x);
```

5 *Requires:* `position` is dereferenceable or equal to `before_begin()`.

6 *Effects:* Inserts a copy of `x` after `position`.

7 *Returns:* An iterator pointing to the copy of `x`.

```
requires AllocatableElement<Alloc, T, const T&>
  void insert_after(const_iterator position, size_type n, const T& x);
```

8 *Requires:* `position` is dereferenceable or equal to `before_begin()`.

9 *Effects:* Inserts `n` copies of `x` after `position`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  void insert_after(const_iterator position, Iter first, Iter last);
```

10 *Requires:* `position` is dereferenceable or equal to `before_begin()`. `first` and `last` are not iterators in `*this`.

11 *Effects:* Inserts copies of elements in `[first,last)` after `position`.

```
requires AllocatableElement<Alloc, T, const T&>
  void insert_after(const_iterator position, initializer_list<T> il);
```

12 *Effects:* `insert_after(p, s.begin(), s.end())`.

```
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  iterator emplace_after(const_iterator position, Args&&... args);
```

13 *Requires:* `position` is dereferenceable or equal to `before_begin()`.

14 *Effects:* Inserts an object of type `value_type` constructed with `value_type(forward<Args>(args)...)` after `position`.

```
iterator erase_after(const_iterator position);
```

15 *Requires:* The iterator following `position` is dereferenceable.

16 *Effects:* Erases the element pointed to by the iterator following `position`.

17 *Returns:* An iterator pointing to the element following the one that was erased, or `end()` if no such element exists.

```
iterator erase_after(const_iterator position, iterator last);
```

18 *Requires:* All iterators in the range `[position,last)` are dereferenceable.

19 *Effects:* Erases the elements in the range `[position,last)`.

20 *Returns:* `last`

```
requires AllocatableElement<Alloc, T> void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, value_type c);
```

21 *Effects:* If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` elements at the end of the list. For the first signature the inserted elements are default constructed, and for the second signature they are copies of `c`.

```
void clear();
```

22 *Effects:* Erases all elements in the range `[begin(),end())`.

23.2.3.5 `forward_list` operations [forwardlist.ops]

```
void splice_after(const_iterator position, forward_list<T,Alloc>&& x);
```

1 *Requires:* `position` is dereferenceable or equal to `before_begin()`. `&x != this`.

2 *Effects:* Inserts the contents of `x` before `position`, and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

3 *Throws:* Nothing.

4 *Complexity:* $\mathcal{O}(1)$

```
void splice_after(const_iterator position, forward_list<T,Alloc>&& x, const_iterator i);
```

5 *Requires:* `position` is dereferenceable or equal to `before_begin()`. The iterator following `i` is a dereferenceable iterator in `x`.

6 *Effects:* Inserts the element following `i` into `*this`, following `position`, and removes it from `x`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

7 *Throws:* Nothing.

8 *Complexity:* $\mathcal{O}(1)$

```
void splice_after(const_iterator position, forward_list<T,Alloc>&& x,
                  const_iterator first, const_iterator last);
```

9 *Requires:* `position` is dereferenceable or equal to `before_begin()`. `(first,last)` is a valid range in `x`, and all iterators in the range `(first,last)` are dereferenceable. `position` is not an iterator in the range `(first,last)`.

10 *Effects:* Inserts elements in the range `(first,last)` after `position` and removes the elements from `x`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

```
requires EqualityComparable<T> void remove(const T& value);
template <Predicate<auto, T> Pred> void remove_if(Pred pred);
```

11 *Effects:* Erases all the elements in the list referred by a list iterator `i` for which the following conditions hold: `*i == value` (for `remove()`), `pred(*i)` is true (for `remove_if()`). This operation shall be stable: the relative order of the elements that are not removed is the same as their relative order in the original list.

12 *Throws:* Nothing unless an exception is thrown by the equality comparison or the predicate.

13 *Complexity:* Exactly `distance(begin(), end())` applications of the corresponding predicate.

```
requires EqualityComparable<T> void unique();
template <EquivalenceRelation<auto, T> BinaryPredicate>
  void unique(BinaryPredicate pred);
```

14 *Effects:* Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1,last)` for which `*i == *(i-1)` (for the version with no arguments) or `pred(*i, *(i - 1))` (for the version with a predicate argument) holds.

15 *Throws:* Nothing unless an exception is thrown by the equality comparison or the predicate.

16 *Complexity:* If the range `[first,last)` is not empty, exactly `(last - first) - 1` applications of the corresponding predicate, otherwise no applications of the predicate.

```
requires LessThanComparable<T> void merge(forward_list<T,Alloc>&& x);
template <StrictWeakOrder<auto, T> Compare>
  void merge(forward_list<T,Alloc>&& x, Compare comp)
```

17 *Requires:* `*this` and `x` are both sorted according to the strict weak ordering defined by `operator<` or `comp`.

18 *Effects:* Merges `x` into `*this`. This operation shall be stable: for equivalent elements in the two lists, the elements from `*this` shall always precede the elements from `x`. `x` is empty after the merge. If an exception is thrown other than by a comparison there are no effects.

19 *Complexity:* At most `size() + x.size() - 1` comparisons.

```
requires LessThanComparable<T> void sort();
template <StrictWeakOrder<auto, T> Compare> void sort(Compare comp);
```

20 *Effects:* Sorts the list according to the `operator<` or the `comp` function object. This operation shall be stable: the relative order of the equivalent elements is preserved. If an exception is thrown the order of the elements in `*this` is unspecified.

21 *Complexity:* Approximately $N \log N$ comparisons, where N is `distance(begin(), end())`.

```
void reverse();
```

22 *Effects:* Reverses the order of the elements in the list.

23 *Throws:* Nothing.

24 *Complexity:* Linear time.

23.2.3.6 `forward_list` specialized algorithms [forwardlist.spec]

```
template <ValueType T, class Alloc>
  void swap(forward_list<T,Alloc>& x, forward_list<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(forward_list<T,Alloc>&& x, forward_list<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(forward_list<T,Alloc>& x, forward_list<T,Alloc>&& y);
```

1 *Effects:* `x.swap(y)`

23.2.4 Class template `list` [list]

1 A `list` is a sequence container that supports bidirectional iterators and allows constant time insert and erase operations anywhere within the sequence, with storage management handled automatically. Unlike vectors (23.2.6) and deques (23.2.2), fast random access to list elements is not supported, but many algorithms only need sequential access anyway.

2 A `list` satisfies all of the requirements of a container, of a reversible container (given in two tables in 23.1), of a sequence container, including most of the the optional sequence container requirements (23.1.3), and of an allocator-aware container (Table 82). The exceptions are the `operator[]` and `at` member functions, which are not provided.[259] Descriptions are provided here only for operations on `list` that are not described in one of these tables or for operations where there is additional semantic information.

```
namespace std {
  template <ValueType T, Allocator Alloc = allocator<T> >
  requires NothrowDestructible<T>
  class list {
  public:
    // types:
    typedef typename Alloc::reference         reference;
    typedef typename Alloc::const_reference   const_reference;
    typedef implementation-defined                iterator;       // See 23.1
    typedef implementation-defined                const_iterator; // See 23.1
    typedef implementation-defined                size_type;      // See 23.1
    typedef implementation-defined                difference_type;// See 23.1
    typedef T                                     value_type;
    typedef Alloc                                 allocator_type;
    typedef typename Alloc::pointer           pointer;
    typedef typename Alloc::const_pointer     const_pointer;
    typedef reverse_iterator<iterator>       reverse_iterator;
    typedef reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.2.4.1 construct/copy/destroy:
    explicit list(const Alloc& = Alloc());
    requires AllocatableElement<Alloc, T> explicit list(size_type n);
    requires AllocatableElement<Alloc, T, const T&>
      list(size_type n, const T& value, const Alloc& = Alloc());
    template <InputIterator Iter>
      requires AllocatableElement<Alloc, T, Iter::reference>
      list(Iter first, Iter last, const Alloc& = Alloc());
    requires AllocatableElement<Alloc, T, const T&> list(const list<T,Alloc>& x);
    requires AllocatableElement<Alloc, T, T&&> list(list&& x);
```

259) These member functions are only provided by containers whose iterators are random access iterators.

```
  requires AllocatableElement<Alloc, T, const T&> list(const list&, const Alloc&);
  requires AllocatableElement<Alloc, T, T&&> list(list&&, const Alloc&);
  requires AllocatableElement<Alloc, T, const T&>
    list(initializer_list<T>, const Allocator& = Allocator());
 ~list();
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    list<T,Alloc>& operator=(const list<T,Alloc>& x);
  requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
    list<T,Alloc>& operator=(list<T,Alloc>&& x);
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    list<T,Alloc>& operator=(initializer_list<T>);
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
          && HasAssign<T, Iter::reference>
    void assign(Iter first, Iter last);
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(size_type n, const T& t);
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(initializer_list<T>);
  allocator_type get_allocator() const;

  // iterators:
  iterator               begin();
  const_iterator         begin() const;
  iterator               end();
  const_iterator         end() const;
  reverse_iterator       rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator       rend();
  const_reverse_iterator rend() const;

  const_iterator         cbegin() const;
  const_iterator         cend() const;
  const_reverse_iterator crbegin() const;
  const_reverse_iterator crend() const;

  // 23.2.4.2 capacity:
  bool      empty() const;
  size_type size() const;
  size_type max_size() const;
  requires AllocatableElement<Alloc, T> void resize(size_type sz);
  requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, const T& c);

  // element access:
  reference       front();
  const_reference front() const;
  reference       back();
  const_reference back() const;

  // 23.2.4.3 modifiers:
  template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_front(Args&&... args);
  void pop_front();
  template <class... Args>
```

```
    requires AllocatableElement<Alloc, T, Args&&...>
    void emplace_back(Args&&... args);
  void pop_back();

  requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
  requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
  requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
  requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);

  template <class... Args>
    requires AllocatableElement<Alloc, T, Args&&...>
    iterator emplace(const_iterator position, Args&&... args);
  requires AllocatableElement<Alloc, T, const T&>
    iterator insert(const_iterator position, const T& x);
  requires AllocatableElement<Alloc, T, T&&>
    iterator insert(const_iterator position, T&& x);
  requires AllocatableElement<Alloc, T, const T&>
    void insert(const_iterator position, size_type n, const T& x);
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
    void insert(const_iterator position, Iter first,
                Iter last);
  requires AllocatableElement<Alloc, T, const T&>
    void insert(const_iterator position, initializer_list<T> il);

  iterator erase(const_iterator position);
  iterator erase(const_iterator position, const_iterator last);
  void     swap(list<T,Alloc>&&);
  void     clear();

  // 23.2.4.4 list operations:
  void splice(const_iterator position, list<T,Alloc>&& x);
  void splice(const_iterator position, list<T,Alloc>&& x, const_iterator i);
  void splice(const_iterator position, list<T,Alloc>&& x,
              const_iterator first, const_iterator last);

  requires EqualityComparable<T> void remove(const T& value);
  template <Predicate<auto, T> Pred> void remove_if(Pred pred);

  requires EqualityComparable<T> void unique();
  template <EquivalenceRelation<auto, T> BinaryPredicate>
    void unique(BinaryPredicate binary_pred);

  requires LessThanComparable<T> void merge(list<T,Alloc>&& x);
  template <StrictWeakOrder<auto, T> Compare>
    void merge(list<T,Alloc>&& x, Compare comp);

  requires LessThanComparable<T> void sort();
  template <StrictWeakOrder<auto, T> Compare>
    void sort(Compare comp);

  void reverse();
};

template <EqualityComparable T, class Alloc>
```

```
    bool operator==(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const list<T,Alloc>& x, const list<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const list<T,Alloc>& x, const list<T,Alloc>& y);

  // specialized algorithms:
  template <ValueType T, class Alloc>
    void swap(list<T,Alloc>& x, list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(list<T,Alloc>&& x, list<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(list<T,Alloc>& x, list<T,Alloc>&& y);
}
```

23.2.4.1 `list` constructors, copy, and assignment [list.cons]

```
explicit list(const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty list, using the specified allocator.

2 *Complexity:* Constant.

```
requires AllocatableElement<Alloc, T> explicit list(size_type n);
```

3 *Effects:* Constructs a `list` with `n` default constructed elements.

4 *Complexity:* Linear in `n`.

```
requires AllocatableElement<Alloc, T, const T&>
  list(size_type n, const T& value,
               const Alloc& = Alloc());
```

5 *Effects:* Constructs a `list` with `n` copies of `value`, using the specified allocator.

6 *Complexity:* Linear in `n`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  list(Iter first, Iter last, const Alloc& = Alloc());
```

7 *Effects:* Constructs a `list` equal to the range `[first,last)`.

8 *Complexity:* Linear in `distance(first, last)`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
        && HasAssign<T, Iter::reference>
  void assign(Iter first, Iter last);
```

9 *Effects:* Replaces the contents of the list with the range `[first, last)`.

```
    erase(begin(), end());
    insert(begin(), n, t);
```

```
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  void assign(size_type n, const T& t);
```

10 *Effects:* Replaces the contents of the list with `n` copies of `t`.

23.2.4.2 `list` capacity [list.capacity]

```
requires AllocatableElement<Alloc, T> void resize(size_type sz);
```

1 *Effects:* If `sz < size()`, equivalent to `list<T>::iterator it = begin(); advance(it, sz); erase(it, end());`. If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

```
requires AllocatableElement<Alloc, T, const T&> void resize(size_type sz, const T& c);
```

2 *Effects:*

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size()) {
  iterator i = begin();
  advance(i, sz);
  erase(i, end());
}
else
  ;                 // do nothing
```

23.2.4.3 `list` modifiers [list.modifiers]

```
requires AllocatableElement<Alloc, T, const T&>
  iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&>
  iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&>
  void insert(const_iterator position, size_type n, const T& x);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  void insert(const_iterator position, Iter first,
              Iter last);

template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_front(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_back(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  iterator emplace(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_front(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_front(T&& x);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);
```

1 *Remarks:* Does not affect the validity of iterators and references. If an exception is thrown there are no effects.

2 *Complexity:* Insertion of a single element into a list takes constant time and exactly one call to a constructor of `T`. Insertion of multiple elements into a list is linear in the number of elements inserted, and the number of calls to the copy constructor or move constructor of `T` is exactly equal to the number of elements inserted.

```
iterator erase(const_iterator position);
iterator erase(const_iterator first, const_iterator last);

void pop_front();
void pop_back();
void clear();
```

3 *Effects:* Invalidates only the iterators and references to the erased elements.

4 *Throws:* Nothing.

5 *Complexity:* Erasing a single element is a constant time operation with a single call to the destructor of `T`. Erasing a range in a list is linear time in the size of the range and the number of calls to the destructor of type `T` is exactly equal to the size of the range.

23.2.4.4 `list` operations [list.ops]

1 Since lists allow fast insertion and erasing from the middle of a list, certain operations are provided specifically for them.[260]

2 `list` provides three splice operations that destructively move elements from one list to another. The behavior of splice operations is undefined if `get_allocator() != x.get_allocator()`.

```
void splice(const_iterator position, list<T,Alloc>&& x);
```

3 *Requires:* `&x != this`.

4 *Effects:* Inserts the contents of `x` before `position` and `x` becomes empty. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

5 *Throws:* Nothing

6 *Complexity:* Constant time.

```
void splice(const_iterator position, list<T,Alloc>&& x, iterator i);
```

7 *Effects:* Inserts an element pointed to by `i` from list `x` before `position` and removes the element from `x`. The result is unchanged if `position == i` or `position == ++i`. Pointers and references to `*i` continue to refer to this same element but as a member of `*this`. Iterators to `*i` (including `i` itself) continue to refer to the same element, but now behave as iterators into `*this`, not into `x`.

8 *Throws:* Nothing

9 *Requires:* `i` is a valid dereferenceable iterator of `x`.

10 *Complexity:* Constant time.

```
void splice(const_iterator position, list<T,Alloc>&& x, iterator first,
            iterator last);
```

260) As specified in 20.7.2.2, the requirements in this clause apply only to lists whose allocators compare equal.

11 *Effects:* Inserts elements in the range `[first,last)` before `position` and removes the elements from `x`.

12 *Requires:* `[first, last)` is a valid range in `x`. The result is undefined if `position` is an iterator in the range `[first,last)`. Pointers and references to the moved elements of `x` now refer to those same elements but as members of `*this`. Iterators referring to the moved elements will continue to refer to their elements, but they now behave as iterators into `*this`, not into `x`.

13 *Throws:* Nothing

14 *Complexity:* Constant time if `&x == this`; otherwise, linear time.

```
requires EqualityComparable<T> void remove(const T& value);
template <Predicate<auto, T> Pred> void remove_if(Pred pred);
```

15 *Effects:* Erases all the elements in the list referred by a list iterator `i` for which the following conditions hold: `*i == value, pred(*i) != false`.

16 *Throws:* Nothing unless an exception is thrown by `*i == value` or `pred(*i) != false`.

17 *Remarks:* Stable.

18 *Complexity:* Exactly `size()` applications of the corresponding predicate.

```
requires EqualityComparable<T> void unique();
template <EquivalenceRelation<auto, T> BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

19 *Effects:* Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first + 1,last)` for which `*i == *(i-1)` (for the version of `unique` with no arguments) or `pred(*i, *(i - 1))` (for the version of `unique` with a predicate argument) holds.

20 *Throws:* Nothing unless an exception in thrown by `*i == *(i-1)` or `pred(*i, *(i - 1))`

21 *Complexity:* If the range `[first, last)` is not empty, exactly `(last - first) - 1` applications of the corresponding predicate, otherwise no applications of the predicate.

```
requires LessThanComparable<T> void merge(list<T,Alloc>&& x);
template <StrictWeakOrder<auto, T> Compare>
  void merge(list<T,Alloc>&& x, Compare comp);
```

22 *Requires:* both the list and the argument list shall be sorted according to `operator<` or `comp`.

23 *Effects:* If `(&x == this)` does nothing; otherwise, merges the two sorted ranges `[begin(), end())` and `[x.begin(), x.end())`. The result is a range in which the elements will be sorted in non-decreasing order according to the ordering defined by `comp`; that is, for every iterator `i`, in the range other than the first, the condition `comp(*i, *(i - 1)` will be false.

24 *Remarks:* Stable. If `(&x != this)` the range `[x.begin(), x.end())` is empty after the merge.

25 *Complexity:* At most `size() + x.size() - 1` applications of `comp` if `(&x != this)`; otherwise, no applications of `comp` are performed. If an exception is thrown other than by a comparison there are no effects.

```
void reverse();
```

26 *Effects:* Reverses the order of the elements in the list.

27 *Throws:* Nothing.

28 *Complexity:* Linear time.

```
requires LessThanComparable<T> void sort();
template <StrictWeakOrder<auto, T> Compare> void sort(Compare comp);
```

29 *Effects:* Sorts the list according to the `operator<` or a `Compare` function object.

30 *Remarks:* Stable.

31 *Complexity:* Approximately $N \log(N)$ comparisons, where `N == size()`.

23.2.4.5 `list` specialized algorithms [list.special]

```
template <ValueType T, class Alloc>
  void swap(list<T,Alloc>& x, list<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(list<T,Alloc>&& x, list<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(list<T,Alloc>& x, list<T,Alloc>&& y);
```

1 *Effects:*

```
x.swap(y);
```

23.2.5 Container adaptors [container.adaptors]

1 The container adaptors each take a `Container` template parameter, and each constructor takes a `Container` reference argument. This container is copied into the `Container` member of each adaptor. If the container takes an allocator, then a compatible allocator may be passed in to the adaptor's constructor. Otherwise, normal copy or move construction is used for the container argument. [*Note:* it is not necessary for an implementation to distinguish between the one-argument constructor that takes a `Container` and the one-argument constructor that takes an `allocator_type`. Both forms use their argument to construct an instance of the container. — *end note*]

23.2.5.1 Class template `queue` [queue]

1 Any sequence container meeting the requirements of the `FrontInsertionContainer` and `BackInsertionContainer` concepts can be used to instantiate `queue`. In particular, `list` (23.2.4) and `deque` (23.2.2) can be used.

23.2.5.1.1 `queue` definition [queue.defn]

```
namespace std {
  template <ObjectType T, class Cont = deque<T> >
  requires QueueLikeContainer<Cont>
        && SameType<T, Cont::value_type>
        && NothrowDestructible<Cont>
  class queue {
  public:
    typedef typename Cont::value_type            value_type;
    typedef typename Cont::reference             reference;
    typedef typename Cont::const_reference       const_reference;
    typedef typename Cont::size_type             size_type;
    typedef          Cont                        container_type;
  protected:
    Cont c;

  public:
    requires CopyConstructible<Cont> explicit queue(const Cont&);
```

```
  requires MoveConstructible<Cont> explicit queue(Cont&& = Cont());
  requires MoveConstructible<Cont> queue(queue&& q) : c(move(q.c)) {}
  template <class Alloc>
    requires Constructible<Cont, const Alloc&>
    explicit queue(const Alloc&);
  template <class Alloc>
    requires Constructible<Cont, const Cont&, const Alloc&>
    queue(const Cont&, const Alloc&);
  template <class Alloc>
    requires Constructible<Cont, Cont&&, const Alloc&>
    queue(Cont&&, const Alloc&);
  template <class Alloc>
    requires Constructible<Cont, Cont&&, const Alloc&>
    queue(queue&&, const Alloc&);
  requires MoveAssignable<Cont> queue& operator=(queue&& q)
                                    { c = move(q.c); return *this; }

  bool              empty() const     { return empty(c); }
  size_type         size()  const     { return size(c); }
  reference         front()           { return front(c); }
  const_reference   front() const     { return front(c); }
  reference         back()            { return back(c); }
  const_reference   back() const      { return back(c); }
  void push(const value_type& x)      { push_back(c, x); }
  void push(value_type&& x)           { push_back(c, move(x)); }
  template <class... Args>
    requires BackEmplacementContainer<Cont, Args&&...>
    void emplace(Args&&... args)
    { emplace_back(c, forward<Args>(args)...); }
  void pop()                          { pop_front(c); }
  requires Swappable<Cont>
    void swap(queue&& q)              { swap(c, q.c); }
};

template <class T, EqualityComparable Cont>
  bool operator==(const queue<T, Cont>& x, const queue<T, Cont>& y);
template <class T, LessThanComparable Cont>
  bool operator< (const queue<T, Cont>& x, const queue<T, Cont>& y);
template <class T, EqualityComparable Cont>
  bool operator!=(const queue<T, Cont>& x, const queue<T, Cont>& y);
template <class T, LessThanComparable Cont>
  bool operator> (const queue<T, Cont>& x, const queue<T, Cont>& y);
template <class T, LessThanComparable Cont>
  bool operator>=(const queue<T, Cont>& x, const queue<T, Cont>& y);
template <class T, LessThanComparable Cont>
  bool operator<=(const queue<T, Cont>& x, const queue<T, Cont>& y);

template <ObjectType T, Swappable Cont>
  void swap(queue<T, Cont>& x, queue<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(queue<T, Cont>&& x, queue<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(queue<T, Cont>& x, queue<T, Cont>&& y);

template <class T, class Cont, class Alloc>
```

```
    requires UsesAllocator<Cont, Alloc>
    concept_map UsesAllocator<queue<T, Cont>, Alloc> { }
}
```

23.2.5.1.2 queue operators [queue.ops]

```
template <class T, EqualityComparable Cont>
    bool operator==(const queue<T, Cont>& x,
                    const queue<T, Cont>& y);
```

1 *Returns:* `x.c == y.c`.

```
template <class T, EqualityComparable Cont>
    bool operator!=(const queue<T, Cont>& x,
                    const queue<T, Cont>& y);
```

2 *Returns:* `x.c != y.c`.

```
template <class T, LessThanComparable Cont>
    bool operator< (const queue<T, Cont>& x,
                    const queue<T, Cont>& y);
```

3 *Returns:* `x.c < y.c`.

```
template <class T, LessThanComparable Cont>
    bool operator<=(const queue<T, Cont>& x,
                    const queue<T, Cont>& y);
```

4 *Returns:* `x.c <= y.c`.

```
template <class T, LessThanComparable Cont>
    bool operator> (const queue<T, Cont>& x,
                    const queue<T, Cont>& y);
```

5 *Returns:* `x.c > y.c`.

```
template <class T, LessThanComparable Cont>
    bool operator>=(const queue<T, Cont>& x,
                    const queue<T, Cont>& y);
```

6 *Returns:* `x.c >= y.c`.

23.2.5.1.3 queue specialized algorithms [queue.special]

```
template <ObjectType T, Swappable Cont>
  void swap(queue<T, Cont>& x, queue<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(queue<T, Cont>&& x, queue<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(queue<T, Cont>& x, queue<T, Cont>&& y);
```

1 *Effects:* `x.swap(y)`.

23.2.5.2 Class template priority_queue [priority.queue]

1 Any sequence container with random access iterator and that meets the requirements of the `BackInsertionContainer` concept can be used to instantiate `priority_queue`. In particular, `vector` (23.2.6) and `deque` (23.2.2) can

be used. Instantiating `priority_queue` also involves supplying a function or function object for making priority comparisons; the library assumes that the function or function object defines a strict weak ordering (25.3).

```
namespace std {
  template <ObjectType T, StackLikeContainer Cont = vector<T>,
            StrictWeakOrder<auto, T> Compare = less<typename Cont::value_type> >
  requires SameType<Cont::value_type, T> && RandomAccessIterator<Cont::iterator>
        && ShuffleIterator<Cont::iterator> && CopyConstructible<Compare>
        && NothrowDestructible<Cont>
  class priority_queue {
  public:
    typedef typename Cont::value_type            value_type;
    typedef typename Cont::reference             reference;
    typedef typename Cont::const_reference       const_reference;
    typedef typename Cont::size_type             size_type;
    typedef          Cont                        container_type;
  protected:
    Cont c;
    Compare comp;

  public:
    requires CopyConstructible<Cont> priority_queue(const Compare& x, const Cont&);
    requires MoveConstructible<Cont>
      explicit priority_queue(const Compare& x = Compare(), Cont&& = Cont());
    template <InputIterator Iter>
      requires CopyConstructible<Cont> && RangeInsertionContainer<Cont, Iter>
      priority_queue(Iter first, Iter last,
             const Compare& x, const Cont&);
    template <InputIterator Iter>
      requires MoveConstructible<Cont> && RangeInsertionContainer<Cont, Iter>
      priority_queue(Iter first, Iter last,
             const Compare& x = Compare(), Cont&& = Cont());
    requires MoveConstructible<Cont> priority_queue(priority_queue&&);
    requires MoveAssignable<Cont> priority_queue& operator=(priority_queue&&);
    template <class Alloc>
      requires Constructible<Cont, const Alloc&>
      explicit priority_queue(const Alloc&);
    template <class Alloc>
      requires Constructible<Cont, const Alloc&>
      priority_queue(const Compare&, const Alloc&);
    template <class Alloc>
      requires Constructible<Cont, Cont, Alloc>
      priority_queue(const Compare&, const Cont&, const Alloc&);
    template <class Alloc>
      requires Constructible<Cont, Cont&&, Alloc>
      priority_queue(const Compare&, Cont&&, const Alloc&);
    template <class Alloc>
      requires Constructible<Cont, Cont&&, Alloc>
      priority_queue(priority_queue&&, const Alloc&);

    bool      empty() const       { return empty(c); }
    size_type size()  const       { return size(c); }
    const_reference   top() const { return *begin(c)); }
    void push(const value_type& x);
    void push(value_type&& x);
```

```
    template <class... Args>
      requires BackEmplacementContainer<Cont, Args&&...>
      void emplace(Args&&... args);
    void pop();
    requires Swappable<Cont>
      void swap(priority_queue&&);
  };
                // no equality is provided
  template <ObjectType T, Swappable Cont, Swappable Compare>
    void swap(priority_queue<T, Cont, Compare>& x, priority_queue<T, Cont, Compare>& y);
  template <ObjectType T, Swappable Cont, Swappable Compare>
    void swap(priority_queue<T, Cont, Compare>&& x, priority_queue<T, Cont, Compare>& y);
  template <ObjectType T, Swappable Cont, Swappable Compare>
    void swap(priority_queue<T, Cont, Compare>& x, priority_queue<T, Cont, Compare>&& y);

  template <class T, class Cont, class Compare, class Alloc>
    requires UsesAllocator<Cont, Alloc>
    concept_map UsesAllocator<priority_queue<T, Cont, Compare>, Alloc> { }
}
```

23.2.5.2.1 `priority_queue` constructors [priqueue.cons]

```
requires CopyConstructible<Cont> priority_queue(const Compare& x, const Cont& y);

requires MoveConstructible<Cont>
explicit priority_queue(const Compare& x = Compare(), Cont&& y = Cont());
```

1 *Effects:* Initializes `comp` with `x` and `c` with `y` (copy constructing or move constructing as appropriate); calls `make_heap(begin(c), end(c), comp)`.

```
template <InputIterator Iter>
  requires CopyConstructible<Cont> && RangeInsertionContainer<Cont, Iter>
  priority_queue(Iter first, Iter last,
         const Compare& x, const Cont&);
template <InputIterator Iter>
  requires MoveConstructible<Cont> && RangeInsertionContainer<Cont, Iter>
  priority_queue(Iter first, Iter last,
         const Compare& x = Compare(), Cont&& = Cont());
```

2 *Effects:* Initializes `comp` with `x` and `c` with `y` (copy constructing or move constructing as appropriate); calls `insert(c, end(c), first, last)`; and finally calls `make_heap(begin(c), end(c), comp)`.

23.2.5.2.2 `priority_queue` members [priqueue.members]

```
void push(const value_type& x);
```

1 *Effects:*

```
  push_back(c, x);
  push_heap(begin(c), end(c), comp);
```

```
void push(value_type&& x);
```

2 *Effects:*

```
  push_back(c, move(x));
  push_heap(begin(c), end(c), comp);
```

```
template <class... Args>
  requires BackEmplacementContainer<Cont, Args&&...>
  void emplace(Args&&... args);
```

3 *Effects:*

```
  emplace_back(c, forward<Args>(args)...);
  push_heap(begin(c), end(c), comp);
```

```
void pop();
```

4 *Effects:*

```
  pop_heap(begin(c), end(c), comp);
  pop_back(c);
```

23.2.5.2.3 `priority_queue` specialized algorithms [priqueue.special]

```
template <class T, Swappable Cont, Swappable Compare>
  void swap(priority_queue<T, Cont, Compare>& x, priority_queue<T, Cont, Compare>& y);
template <class T, Swappable Cont, Swappable Compare>
  void swap(priority_queue<T, Cont, Compare>&& x, priority_queue<T, Cont, Compare>& y);
template <class T, Swappable Cont, Swappable Compare>
  void swap(priority_queue<T, Cont, Compare>& x, priority_queue<T, Cont, Compare>&& y);
```

1 *Effects:* `x.swap(y)`.

23.2.5.3 Class template `stack` [stack]

1 Any sequence container that meets the requirements of the `BackInsertionContainer` concept can be used to instantiate `stack`. In particular, `vector` (23.2.6), `list` (23.2.4) and `deque` (23.2.2) can be used.

23.2.5.3.1 `stack` definition [stack.defn]

```
namespace std {
  template <ObjectType T, StackLikeContainer Cont = deque<T> >
  requires SameType<Cont::value_type, T>
        && NothrowDestructible<Cont>
  class stack {
  public:
    typedef typename Cont::value_type            value_type;
    typedef typename Cont::reference             reference;
    typedef typename Cont::const_reference       const_reference;
    typedef typename Cont::size_type             size_type;
    typedef          Cont                        container_type;
  protected:
    Cont c;

  public:
    requires CopyConstructible<Cont> explicit stack(const Cont&);
    requires MoveConstructible<Cont> explicit stack(Cont&& = Cont());
    template <class Alloc>
      requires Constructible<Cont, const Alloc&>
      explicit stack(const Alloc&);
    template <class Alloc>
      requires Constructible<Cont, const Cont&, const Alloc&>
```

```
    stack(const Cont&, const Alloc&);
  template <class Alloc>
    requires Constructible<Cont, Cont&&, const Alloc&>
    stack(Cont&&, const Alloc&);
  template <class Alloc>
    Constructible<Cont, Cont&&, const Alloc&>
    stack(stack&&, const Alloc&);

  bool      empty() const             { return empty(c); }
  size_type size()  const             { return size(c); }
  reference         top()             { return back(c); }
  const_reference   top() const       { return back(c); }
  void push(const value_type& x)      { push_back(c, x); }
  void push(value_type&& x)           { push_back(c, move(x)); }
  template <class... Args>
    requires BackEmplacementContainer<Cont, Args&&...>
    void emplace(Args&&... args)
    { emplace_back(c, forward<Args>(args)...); }
  void pop()                          { pop_back(c); }
  requires Swappable<Cont>
    void swap(stack&& s)              { swap(c, s.c); }
};

template <EqualityComparable T, class Cont>
  bool operator==(const stack<T, Cont>& x, const stack<T, Cont>& y);
template <LessThanComparable T, class Cont>
  bool operator< (const stack<T, Cont>& x, const stack<T, Cont>& y);
template <EqualityComparable T, class Cont>
  bool operator!=(const stack<T, Cont>& x, const stack<T, Cont>& y);
template <LessThanComparable T, class Cont>
  bool operator> (const stack<T, Cont>& x, const stack<T, Cont>& y);
template <LessThanComparable T, class Cont>
  bool operator>=(const stack<T, Cont>& x, const stack<T, Cont>& y);
template <LessThanComparable T, class Cont>
  bool operator<=(const stack<T, Cont>& x, const stack<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(stack<T, Cont>& x, stack<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(stack<T, Cont>&& x, stack<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(stack<T, Cont>& x, stack<T, Cont>&& y);

template <class T, class Cont, class Alloc>
  requires UsesAllocator<Cont, Alloc>
  concept_map UsesAllocator<stack<T, Cont>, Alloc> { }
}
```

23.2.5.3.2 stack operators [stack.ops]

```
template <EqualityComparable T, class Cont>
    bool operator==(const stack<T, Cont>& x,
                    const stack<T, Cont>& y);
```

1 *Returns:* `x.c == y.c`.

```
template <EqualityComparable T, class Cont>
```

```
    bool operator!=(const stack<T, Cont>& x,
                    const stack<T, Cont>& y);
```

2 *Returns:* `x.c != y.c`.

```
template <LessThanComparable T, class Cont>
    bool operator< (const stack<T, Cont>& x,
                    const stack<T, Cont>& y);
```

3 *Returns:* `x.c < y.c`.

```
template <LessThanComparable T, class Cont>
    bool operator<=(const stack<T, Cont>& x,
                    const stack<T, Cont>& y);
```

4 *Returns:* `x.c <= y.c`.

```
template <LessThanComparable T, class Cont>
    bool operator> (const stack<T, Cont>& x,
                    const stack<T, Cont>& y);
```

5 *Returns:* `x.c > y.c`.

```
template <LessThanComparable T, class Cont>
    bool operator>=(const stack<T, Cont>& x,
                    const stack<T, Cont>& y);
```

6 *Returns:* `x.c >= y.c`.

23.2.5.3.3 `stack` specialized algorithms [stack.special]

```
template <ObjectType T, Swappable Cont>
  void swap(stack<T, Cont>& x, stack<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(stack<T, Cont>&& x, stack<T, Cont>& y);
template <ObjectType T, Swappable Cont>
  void swap(stack<T, Cont>& x, stack<T, Cont>&& y);
```

1 *Effects:* `x.swap(y)`.

23.2.6 Class template `vector` [vector]

1 A `vector` is a sequence container that supports random access iterators. In addition, it supports (amortized) constant time insert and erase operations at the end; insert and erase in the middle take linear time. Storage management is handled automatically, though hints can be given to improve efficiency. The elements of a vector are stored contiguously, meaning that if `v` is a `vector<T, Alloc>` where `T` is some type other than `bool`, then it obeys the identity `&v[n] == &v[0] + n` for all `0 <= n < v.size()`.

2 A `vector` satisfies all of the requirements of a container and of a reversible container (given in two tables in 23.1), of a sequence container, including most of the optional sequence container requirements (23.1.3), and of an allocator-aware container (Table 82). The exceptions are the `push_front` and `pop_front` member functions, which are not provided. Descriptions are provided here only for operations on `vector` that are not described in one of these tables or for operations where there is additional semantic information.

```
  namespace std {
    template <ValueType T, Allocator Alloc = allocator<T> >
    requires MoveConstructible<T>
```

```
class vector {
public:
  // types:
  typedef typename Alloc::reference         reference;
  typedef typename Alloc::const_reference   const_reference;
  typedef implementation-defined                iterator;       // See 23.1
  typedef implementation-defined                const_iterator; // See 23.1
  typedef implementation-defined                size_type;      // See 23.1
  typedef implementation-defined                difference_type;// See 23.1
  typedef T                                     value_type;
  typedef Alloc                                 allocator_type;
  typedef typename Alloc::pointer           pointer;
  typedef typename Alloc::const_pointer     const_pointer;
  typedef reverse_iterator<iterator>       reverse_iterator;
  typedef reverse_iterator<const_iterator> const_reverse_iterator;

  // 23.2.6.1 construct/copy/destroy:
  explicit vector(const Alloc& = Alloc());
  requires AllocatableElement<Alloc, T>
    explicit vector(size_type n);
  requires AllocatableElement<Alloc, T, const T&>
    vector(size_type n, const T& value, const Alloc& = Alloc());
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
    vector(Iter first, Iter last,
           const Alloc& = Alloc());
  requires AllocatableElement<Alloc, T, const T&> vector(const vector<T,Alloc>& x);
  requires AllocatableElement<Alloc, T, T&&> vector(vector&&);
  requires AllocatableElement<Alloc, T, const T&> vector(const vector&, const Alloc&);
  requires AllocatableElement<Alloc, T, T&&> vector(vector&&, const Alloc&);
  requires AllocatableElement<Alloc, T, const T&>
    vector(initializer_list<T>, const Allocator& = Allocator());
 ~vector();
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    vector<T,Alloc>& operator=(const vector<T,Alloc>& x);
  requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
    vector<T,Alloc>& operator=(vector<T,Alloc>&& x);
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    vector<T,Alloc>& operator=(initializer_list<T>);
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
          && HasAssign<T, Iter::reference>
    void assign(Iter first, Iter last);
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(size_type n, const T& u);
  requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
    void assign(initializer_list<T>);
  allocator_type get_allocator() const;

  // iterators:
  iterator               begin();
  const_iterator         begin() const;
  iterator               end();
  const_iterator         end() const;
  reverse_iterator       rbegin();
```

```
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// 23.2.6.2 capacity:
size_type size() const;
size_type max_size() const;
requires AllocatableElement<Alloc, T>
  void resize(size_type sz);
requires AllocatableElement<Alloc, T, const T&>
  void resize(size_type sz, const T& c);
size_type capacity() const;
bool      empty() const;
void reserve(size_type n);
void shrink_to_fit();

// element access:
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference       at(size_type n);
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;

// 23.2.6.3 data access
pointer         data();
const_pointer   data() const;

// 23.2.6.4 modifiers:
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_back(Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);
void pop_back();

template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
        && MoveAssignable<T>
  iterator emplace(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
  void insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  void insert(const_iterator position, size_type n, const T& x);
template <InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
```

```
            && MoveAssignable<T>
        void insert(const_iterator position,
                    Iter first, Iter last);
    requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
      void insert(const_iterator position, initializer_list<T> il);
    requires MoveAssignable<T> iterator erase(const_iterator position);
    requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
    void     swap(vector<T,Alloc>&&);
    void     clear();
  };

  template <EqualityComparable T, class Alloc>
    bool operator==(const vector<T,Alloc>& x, const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator< (const vector<T,Alloc>& x, const vector<T,Alloc>& y);
  template <EqualityComparable T, class Alloc>
    bool operator!=(const vector<T,Alloc>& x, const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator> (const vector<T,Alloc>& x, const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator>=(const vector<T,Alloc>& x, const vector<T,Alloc>& y);
  template <LessThanComparable T, class Alloc>
    bool operator<=(const vector<T,Alloc>& x, const vector<T,Alloc>& y);

  // specialized algorithms:
  template <ValueType T, class Alloc>
    void swap(vector<T,Alloc>& x, vector<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(vector<T,Alloc>&& x, vector<T,Alloc>& y);
  template <ValueType T, class Alloc>
    void swap(vector<T,Alloc>& x, vector<T,Alloc>&& y);
}
```

23.2.6.1 `vector` constructors, copy, and assignment [vector.cons]

```
vector(const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty `vector`, using the specified allocator.

2 *Complexity:* Constant.

```
requires AllocatableElement<Alloc, T> explicit vector(size_type n);
```

3 *Effects:* Constructs a `vector` with `n` default constructed elements.

4 *Complexity:* Linear in `n`.

```
requires AllocatableElement<Alloc, T, const T&>
explicit vector(size_type n, const T& value,
                const Alloc& = Alloc());
```

5 *Effects:* Constructs a `vector` with `n` copies of `value`, using the specified allocator.

6 *Complexity:* Linear in `n`.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
  vector(Iter first, Iter last,
```

```
    const Alloc& = Alloc());
```

7 *Effects:* Constructs a `vector` equal to the range `[first,last)`, using the specified allocator.

8 *Complexity:* Makes only N calls to the copy constructor of `T` (where N is the distance between `first` and `last`) and no reallocations if `Iter` meets the requirements of the `ForwardIterator` concept. It makes order `N` calls to the copy constructor of `T` and order $\log(N)$ reallocations if they are just input iterators.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, T, Iter::reference>
        && HasAssign<T, Iter::reference>
  void assign(Iter first, Iter last);
```

9 *Effects:*

```
  erase(begin(), end());
  insert(begin(), first, last);
```

```
requires AllocatableElement<Alloc, T, const T&> && CopyAssignable<T>
  void assign(size_type n, const T& t);
```

10 *Effects:*

```
  erase(begin(), end());
  insert(begin(), n, t);
```

23.2.6.2 `vector` capacity [vector.capacity]

```
size_type capacity() const;
```

1 *Returns:* The total number of elements that the vector can hold without requiring reallocation.

```
void reserve(size_type n);
```

2 *Requires:* If `value_type` has a move constructor, that constructor shall not throw any exceptions.

3 *Effects:* A directive that informs a `vector` of a planned change in size, so that it can manage the storage allocation accordingly. After `reserve()`, `capacity()` is greater or equal to the argument of `reserve` if reallocation happens; and equal to the previous value of `capacity()` otherwise. Reallocation happens at this point if and only if the current capacity is less than the argument of `reserve()`. If an exception is thrown, there are no effects.

4 *Complexity:* It does not change the size of the sequence and takes at most linear time in the size of the sequence.

5 *Throws:* `length_error` if `n > max_size()`.[261]

6 *Remarks:* Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size of the vector greater than the value of `capacity()`.

```
void shrink_to_fit();
```

7 *Remarks:* `shrink_to_fint` is a non-binding request to reduce `capacity()` to `size()`. [*Note:* The request is non-binding to allow latitude for implementation-specific optimizations. — *end note*]

261) `reserve()` uses `Alloc::allocate()` which may throw an appropriate exception.

```
void swap(vector<T,Alloc>&& x);
```

8 *Effects:* Exchanges the contents and `capacity()` of `*this` with that of `x`.

9 *Complexity:* Constant time.

```
requires AllocatableElement<Alloc, T>
  void resize(size_type sz);
```

10 *Effects:* If `sz < size()`, equivalent to `erase(begin() + sz, end());`. If `size() < sz`, appends `sz - size()` default constructed elements to the sequence.

```
requires AllocatableElement<Alloc, T, const T&>
  void resize(size_type sz, const T& c);
```

11 *Effects:*

```
if (sz > size())
  insert(end(), sz-size(), c);
else if (sz < size())
  erase(begin()+sz, end());
else
  ;                 // do nothing
```

12 *Requires:* If `value_type` has a move constructor, that constructor shall not throw any exceptions.

23.2.6.3 `vector` data [vector.data]

```
pointer         data();
const_pointer   data() const;
```

1 *Returns:* A pointer such that `[data(),data() + size())` is a valid range. For a non-empty vector, `data() == &front()`.

2 *Complexity:* Constant time.

3 *Throws:* Nothing.

23.2.6.4 `vector` modifiers [vector.modifiers]

```
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  iterator insert(const_iterator position, const T& x);
requires AllocatableElement<Alloc, T, T&&> && MoveAssignable<T>
  iterator insert(const_iterator position, T&& x);
requires AllocatableElement<Alloc, T, const T&> && MoveAssignable<T>
  void insert(const_iterator position, size_type n, const T& x);
template <InputIterator Iter>
    requires AllocatableElement<Alloc, T, Iter::reference>
          && MoveAssignable<T>
    void insert(const_iterator position,
                Iter first, Iter last);

template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
  void emplace_back(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, T, Args&&...>
```

```
        && MoveAssignable<T>
  iterator emplace(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, T, const T&> void push_back(const T& x);
requires AllocatableElement<Alloc, T, T&&>      void push_back(T&& x);
```

1 *Requires:* If `value_type` has a move constructor, that constructor shall not throw any exceptions.

2 *Remarks:* Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. If an exception is thrown other than by the copy constructor or assignment operator of `T` or by any `InputIterator` operation there are no effects.

3 *Complexity:* The complexity is linear in the number of elements inserted plus the distance to the end of the vector.

```
requires MoveAssignable<T> iterator erase(const_iterator position);
requires MoveAssignable<T> iterator erase(const_iterator first, const_iterator last);
```

4 *Effects:* Invalidates iterators and references at or after the point of the erase.

5 *Complexity:* The destructor of `T` is called the number of times equal to the number of the elements erased, but the move assignment operator of `T` is called the number of times equal to the number of elements in the vector after the erased elements.

6 *Throws:* Nothing unless an exception is thrown by the copy constructor or assignment operator of `T`.

23.2.6.5 `vector` specialized algorithms [vector.special]

```
template <ValueType T, class Alloc>
  void swap(vector<T,Alloc>& x, vector<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(vector<T,Alloc>&& x, vector<T,Alloc>& y);
template <ValueType T, class Alloc>
  void swap(vector<T,Alloc>& x, vector<T,Alloc>&& y);
```

1 *Effects:*

```
  x.swap(y);
```

23.2.7 Class `vector<bool>` [vector.bool]

1 To optimize space allocation, a specialization of vector for `bool` elements is provided:

```
namespace std {
  template <Allocator Alloc> class vector<bool, Alloc> {
  public:
    // types:
    typedef bool                                 const_reference;
    typedef implementation-defined               iterator;       // See 23.1
    typedef implementation-defined               const_iterator; // See 23.1
    typedef implementation-defined               size_type;      // See 23.1
    typedef implementation-defined               difference_type;// See 23.1
    typedef bool                                 value_type;
    typedef Alloc                               allocator_type;
    typedef implementation-defined               pointer;
    typedef implementation-defined               const_pointer;
    typedef reverse_iterator<iterator>      reverse_iterator;
```

```
typedef reverse_iterator<const_iterator> const_reverse_iterator;

// bit reference:
class reference {
  friend class vector;
  reference();
public:
  ~reference();
  operator bool() const;
  reference& operator=(const bool x);
  reference& operator=(const reference& x);
  void flip();              // flips the bit
};

// construct/copy/destroy:
explicit vector(const Alloc& = Alloc());
explicit vector(size_type n, const bool& value = bool(),
                const Alloc& = Alloc());
template <InputIterator Iter>
  requires Convertible<Iter::reference, bool>
  vector(Iter first, Iter last,
         const Alloc& = Alloc());
vector(const vector<bool,Alloc>& x);
vector(vector<bool,Alloc>&& x);
vector(const vector&, const Alloc&);
vector(vector&&, const Alloc&);
vector(initializer_list<bool>);
~vector();
vector<bool,Alloc>& operator=(const vector<bool,Alloc>& x);
vector<bool,Alloc>& operator=(vector<bool,Alloc>&& x);
vector<bool,Alloc>& operator=(initializer_list<bool>);
template <InputIterator Iter>
  requires Convertible<Iter::reference, bool>
  void assign(Iter first, Iter last);
void assign(size_type n, const bool& t);
void assign(initializer_list<bool>);
allocator_type get_allocator() const;

// iterators:
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
size_type size() const;
```

```
    size_type max_size() const;
    void      resize(size_type sz, bool c = false);
    size_type capacity() const;
    bool      empty() const;
    void      reserve(size_type n);
    void      shrink_to_fit();

    // element access:
    reference       operator[](size_type n);
    const_reference operator[](size_type n) const;
    const_reference at(size_type n) const;
    reference       at(size_type n);
    reference       front();
    const_reference front() const;
    reference       back();
    const_reference back() const;

    // modifiers:
    void push_back(const bool& x);
    void pop_back();
    iterator insert(const_iterator position, const bool& x);
    void     insert (const_iterator position, size_type n, const bool& x);
    template <InputIterator Iter>
        requires Convertible<Iter::reference, bool>
        void insert(const_iterator position,
                    Iter first, Iter last);
    void insert(const_iterator position, initializer_list<bool> il);

    iterator erase(const_iterator position);
    iterator erase(const_iterator first, const_iterator last);
    void swap(vector<bool,Alloc>&&);
    static void swap(reference x, reference y);
    void flip();                // flips all bits
    void clear();
  };

}
```

2 Unless described below, all operations have the same requirements and semantics as the primary `vector` template, except that operations dealing with the `bool` value type map to bit values in the container storage and `construct_element` (23.1) is not used to construct these values.

3 There is no requirement that the data be stored as a contiguous allocation of `bool` values. A space-optimized representation of bits is recommended instead.

4 `reference` is a class that simulates the behavior of references of a single bit in `vector<bool>`. The conversion operator returns `true` when the bit is set, and `false` otherwise. The assignment operator sets the bit when the argument is (convertible to) `true` and clears it otherwise. `flip` reverses the state of the bit.

`void flip();`

5 *Effects:* Replaces each element in the container with its complement. It is unspecified whether the function has any effect on allocated but unused bits.

23.3 Associative containers [associative]

1 Headers <map> and <set>:

Header <map> synopsis

```
namespace std {
  template <ValueType Key, ValueType T,
            Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
    requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
             && AllocatableElement<Alloc, Compare, const Compare&>
             && AllocatableElement<Alloc, Compare, Compare&&>
    class map;
  template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
    bool operator==(const map<Key,T,Compare,Alloc>& x,
                    const map<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator< (const map<Key,T,Compare,Alloc>& x,
                    const map<Key,T,Compare,Alloc>& y);
  template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
    bool operator!=(const map<Key,T,Compare,Alloc>& x,
                    const map<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator> (const map<Key,T,Compare,Alloc>& x,
                    const map<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator>=(const map<Key,T,Compare,Alloc>& x,
                    const map<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator<=(const map<Key,T,Compare,Alloc>& x,
                    const map<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(map<Key,T,Compare,Alloc>& x,
              map<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(map<Key,T,Compare,Alloc&& x,
              map<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(map<Key,T,Compare,Alloc& x,
              map<Key,T,Compare,Alloc>&& y);

  template <ValueType Key, ValueType T,
            Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
    requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
             && AllocatableElement<Alloc, Compare, const Compare&>
             && AllocatableElement<Alloc, Compare, Compare&&>
    class multimap;
  template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
    bool operator==(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator< (const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
```

```
    bool operator!=(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator> (const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator>=(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator<=(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(multimap<Key,T,Compare,Alloc>& x,
              multimap<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(multimap<Key,T,Compare,Alloc&& x,
              multimap<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(multimap<Key,T,Compare,Alloc& x,
              multimap<Key,T,Compare,Alloc>&& y);
}
```

Header <set> synopsis

```
namespace std {
  template <ValueType Key, Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<Key> >
    requires NothrowDestructible<Key> && CopyConstructible<Compare>
             && AllocatableElement<Alloc, Compare, const Compare&>
             && AllocatableElement<Alloc, Compare, Compare&&>
    class set;
  template <EqualityComparable Key, class Compare, class Alloc>
    bool operator==(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator< (const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <EqualityComparable Key, class Compare, class Alloc>
    bool operator!=(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator> (const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator>=(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator<=(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <ValueType Key, class Compare, class Alloc>
    void swap(set<Key,Compare,Alloc>& x,
              set<Key,Compare,Alloc>& y);
  template <ValueType Key, class T, class Compare, class Alloc>
    void swap(set<Key,T,Compare,Alloc&& x,
              set<Key,T,Compare,Alloc>& y);
  template <ValueType Key, class T, class Compare, class Alloc>
```

```
    void swap(set<Key,T,Compare,Alloc& x,
              set<Key,T,Compare,Alloc>&& y);

  template <ValueType Key, Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<Key> >
    requires NothrowDestructible<Key> && CopyConstructible<Compare>
             && AllocatableElement<Alloc, Compare, const Compare&>
             && AllocatableElement<Alloc, Compare, Compare&&>
    class multiset;
  template <EqualityComparable Key, class Compare, class Alloc>
    bool operator==(const multiset<Key,Compare,Alloc>& x,
                    const multiset<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator< (const multiset<Key,Compare,Alloc>& x,
                    const multiset<Key,Compare,Alloc>& y);
  template <EqualityComparable Key, class Compare, class Alloc>
    bool operator!=(const multiset<Key,Compare,Alloc>& x,
                    const multiset<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator> (const multiset<Key,Compare,Alloc>& x,
                    const multiset<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator>=(const multiset<Key,Compare,Alloc>& x,
                    const multiset<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator<=(const multiset<Key,Compare,Alloc>& x,
                    const multiset<Key,Compare,Alloc>& y);
  template <ValueType Key, class Compare, class Alloc>
    void swap(multiset<Key,Compare,Alloc>& x,
              multiset<Key,Compare,Alloc>& y);
  template <ValueType Key, class T, class Compare, class Alloc>
    void swap(multiset<Key,T,Compare,Alloc&& x,
              multiset<Key,T,Compare,Alloc>& y);
  template <ValueType Key, class T, class Compare, class Alloc>
    void swap(multiset<Key,T,Compare,Alloc& x,
              multiset<Key,T,Compare,Alloc>&& y);
}
```

23.3.1 Class template map [map]

1 A map is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of values of another type T based on the keys. The map class supports bidirectional iterators.

2 A map satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container (23.1.4), and of an allocator-aware container (Table 82). A map also provides most operations described in (23.1.4) for unique keys. This means that a map supports the a_uniq operations in (23.1.4) but not the a_eq operations. For a map<Key,T> the key_type is Key and the value_type is pair<const Key,T>. Descriptions are provided here only for operations on map that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <ValueType Key, ValueType T,
            Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
```

```
requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
      && AllocatableElement<Alloc, Compare, const Compare&>
      && AllocatableElement<Alloc, Compare, Compare&&>
class map {
public:
  // types:
  typedef Key                                    key_type;
  typedef T                                      mapped_type;
  typedef pair<const Key, T>                     value_type;
  typedef Compare                                key_compare;
  typedef Alloc                                 allocator_type;
  typedef typename Alloc::reference             reference;
  typedef typename Alloc::const_reference       const_reference;
  typedef implementation-defined                 iterator;       // See 23.1
  typedef implementation-defined                 const_iterator; // See 23.1
  typedef implementation-defined                 size_type;      // See 23.1
  typedef implementation-defined                 difference_type;// See 23.1
  typedef typename Alloc::pointer               pointer;
  typedef typename Alloc::const_pointer         const_pointer;
  typedef reverse_iterator<iterator>           reverse_iterator;
  typedef reverse_iterator<const_iterator> const_reverse_iterator;

  class value_compare
    : public binary_function<value_type,value_type,bool> {
  friend class map;
  protected:
    Compare comp;
    value_compare(Compare c) : comp(c) {}
  public:
    bool operator()(const value_type& x, const value_type& y) const {
      return comp(x.first, y.first);
    }
  };

  // 23.3.1.1 construct/copy/destroy:
  explicit map(const Compare& comp = Compare(),
               const Alloc& = Alloc());
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, value_type, Iter::reference>
          && MoveConstructible<value_type>
    map(Iter first, Iter last,
        const Compare& comp = Compare(), const Alloc& = Alloc());
  requires AllocatableElement<Alloc, value_type, const value_type&>
    map(const map<Key,T,Compare,Alloc>& x);
  map(map<Key,T,Compare,Alloc>&& x);
  map(const Alloc&);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    map(const map&, const Alloc&);
  requires AllocatableElement<Alloc, value_type, value_type&&>
    map(map&&, const Alloc&);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    map(initializer_list<value_type>,
        const Compare& = Compare(),
        const Allocator& = Allocator());
 ~map();
```

```
requires AllocatableElement<Alloc, value_type, const value_type&>
      && CopyAssignable<value_type>
  map<Key,T,Compare,Alloc>& operator=(const map<Key,T,Compare,Alloc>& x);
map<Key,T,Compare,Alloc>&
  operator=(map<Key,T,Compare,Alloc>&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
      && CopyAssignable<value_type>
  map<Key,T,Compare,Alloc>& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// iterators:
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;

reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool      empty() const;
size_type size() const;
size_type max_size() const;

// 23.3.1.2 element access:
requires AllocatableElement<Alloc, value_type, const key_type&, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  T& operator[](const key_type& x);
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  T& operator[](key_type&& x);
T&       at(const key_type& x);
const T& at(const key_type& x) const;

// modifiers:
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
  pair<iterator, bool> insert(const value_type& x);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  pair<iterator, bool> insert(P&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator position, const value_type& x);
```

```
  template <class P>
    requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
    iterator insert(const_iterator position, P&&);
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, value_type, Iter::reference>
          && MoveConstructible<value_type>
    void insert(Iter first, Iter last);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    void insert(initializer_list<value_type>);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(map<Key,T,Compare,Alloc>&&);
  void clear();

  // observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;

  // 23.3.1.4 map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type      count(const key_type& x) const;

  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;

  pair<iterator,iterator>
    equal_range(const key_type& x);
  pair<const_iterator,const_iterator>
    equal_range(const key_type& x) const;
};

template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
  bool operator==(const map<Key,T,Compare,Alloc>& x,
                  const map<Key,T,Compare,Alloc>& y);
template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
  bool operator< (const map<Key,T,Compare,Alloc>& x,
                  const map<Key,T,Compare,Alloc>& y);
template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
  bool operator!=(const map<Key,T,Compare,Alloc>& x,
                  const map<Key,T,Compare,Alloc>& y);
template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
  bool operator> (const map<Key,T,Compare,Alloc>& x,
                  const map<Key,T,Compare,Alloc>& y);
template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
  bool operator>=(const map<Key,T,Compare,Alloc>& x,
                  const map<Key,T,Compare,Alloc>& y);
template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
  bool operator<=(const map<Key,T,Compare,Alloc>& x,
                  const map<Key,T,Compare,Alloc>& y);
```

```
    // specialized algorithms:
    template <ValueType Key, ValueType T, class Compare, class Alloc>
      void swap(map<Key,T,Compare,Alloc>& x,
                map<Key,T,Compare,Alloc>& y);
    template <ValueType Key, ValueType T, class Compare, class Alloc>
      void swap(map<Key,T,Compare,Alloc&& x,
                map<Key,T,Compare,Alloc>& y);
    template <ValueType Key, ValueType T, class Compare, class Alloc>
      void swap(map<Key,T,Compare,Alloc& x,
                map<Key,T,Compare,Alloc>&& y);
  }
```

23.3.1.1 `map` constructors, copy, and assignment [map.cons]

```
explicit map(const Compare& comp = Compare(),
             const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty `map` using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
         && MoveConstructible<value_type>
  map(Iter first, Iter last,
      const Compare& comp = Compare(), const Alloc& = Alloc());
```

3 *Effects:* Constructs an empty `map` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

4 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.3.1.2 `map` element access [map.access]

```
requires AllocatableElement<Alloc, value_type, const key_type&>, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  T& operator[](const key_type& x);
```

1 *Effects:* If there is no key equivalent to `x` in the map, inserts `value_type(x, T())` into the map.

2 *Returns:* A reference to the `mapped_type` corresponding to `x` in `*this`.

3 *Complexity:* logarithmic.

```
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  T& operator[](key_type&& x);
```

4 *Effects:* If there is no key equivalent to `x` in the map, inserts `value_type(move(x), T())` into the map.

5 *Returns:* A reference to the `mapped_type` corresponding to `x` in `*this`.

6 *Complexity:* logarithmic.

```
T&       at(const key_type& x);
const T& at(const key_type& x) const;
```

7 *Returns:* A reference to the element whose key is equivalent to `x`.

8 *Throws:* An exception object of type `out_of_range` if no such element is present.

9 *Complexity:* logarithmic.

23.3.1.3 `map` modifiers [map.modifiers]

```
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  pair<iterator, bool> insert(P&& x);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  pair<iterator, bool> insert(const_iterator position, P&& x);
```

1 If `P` is instantiated as a reference type, then the argument `x` is copied from. Otherwise `x` is considered to be an rvalue as it is converted to `value_type` and inserted into the `map`. Specifically, in such cases `CopyConstructible` is not required of `key_type` or `mapped_type` unless the conversion from `P` specifically requires it (e.g. if `P` is a `tuple<const key_type, mapped_type>`, then `key_type` must be `CopyConstructible`). The signature taking `InputIterator` parameters does not require `CopyConstructible` of either `key_type` or `mapped_type` if the dereferenced `InputIterator` returns a non-const rvalue `pair<key_type,mapped_type>`. Otherwise `CopyConstructible` is required for both `key_type` and `mapped_type`.

23.3.1.4 `map` operations [map.ops]

```
iterator       find(const key_type& x);
const_iterator find(const key_type& x) const;

iterator       lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

iterator       upper_bound(const key_type& x);
const_iterator upper_bound(const key_type &x) const;

pair<iterator, iterator>
  equal_range(const key_type &x);
pair<const_iterator, const_iterator>
  equal_range(const key_type& x) const;
```

1 The `find`, `lower_bound`, `upper_bound` and `equal_range` member functions each have two versions, one const and the other non-const. In each case the behavior of the two functions is identical except that the const version returns a `const_iterator` and the non-const version an `iterator` (23.1.4).

23.3.1.5 `map` specialized algorithms [map.special]

```
template <ValueType Key, ValueType T, class Compare, class Alloc>
  void swap(map<Key,T,Compare,Alloc>& x,
            map<Key,T,Compare,Alloc>& y);
template <ValueType Key, ValueType T, class Compare, class Alloc>
  void swap(map<Key,T,Compare,Alloc>&& x,
            map<Key,T,Compare,Alloc>& y);
template <ValueType Key, ValueType T, class Compare, class Alloc>
  void swap(map<Key,T,Compare,Alloc>& x,
```

```
        map<Key,T,Compare,Alloc>&& y);
```

1 *Effects:*

```
x.swap(y);
```

23.3.2 Class template `multimap` [multimap]

1 A `multimap` is an associative container that supports equivalent keys (possibly containing multiple copies of the same key value) and provides for fast retrieval of values of another type `T` based on the keys. The `multimap` class supports bidirectional iterators.

2 A `multimap` satisfies all of the requirements of a container and of a reversible container (23.1), of an associative container (23.1.4), and of an allocator-aware container (Table 82). A `multimap` also provides most operations described in (23.1.4) for equal keys. This means that a `multimap` supports the `a_eq` operations in (23.1.4) but not the `a_uniq` operations. For a `multimap<Key,T>` the `key_type` is `Key` and the `value_type` is `pair<const Key,T>`. Descriptions are provided here only for operations on `multimap` that are not described in one of those tables or for operations where there is additional semantic information.

```
namespace std {
  template <ValueType Key, ValueType T,
            Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
    requires NothrowDestructible<Key> && NothrowDestructible<T> && CopyConstructible<Compare>
          && AllocatableElement<Alloc, Compare, const Compare&>
          && AllocatableElement<Alloc, Compare, Compare&&>
  class multimap {
  public:
    // types:
    typedef Key                                    key_type;
    typedef T                                      mapped_type;
    typedef pair<const Key,T>                      value_type;
    typedef Compare                                key_compare;
    typedef Alloc                                allocator_type;
    typedef typename Alloc::reference            reference;
    typedef typename Alloc::const_reference      const_reference;
    typedef implementation-defined                   iterator;       // See 23.1
    typedef implementation-defined                   const_iterator; // See 23.1
    typedef implementation-defined                   size_type;      // See 23.1
    typedef implementation-defined                   difference_type;// See 23.1
    typedef typename Alloc::pointer              pointer;
    typedef typename Alloc::const_pointer        const_pointer;
    typedef reverse_iterator<iterator>          reverse_iterator;
    typedef reverse_iterator<const_iterator> const_reverse_iterator;

    class value_compare
      : public binary_function<value_type,value_type,bool> {
    friend class multimap;
    protected:
      Compare comp;
      value_compare(Compare c) : comp(c) { }
    public:
      bool operator()(const value_type& x, const value_type& y) const {
        return comp(x.first, y.first);
      }
```

```
};

// construct/copy/destroy:
explicit multimap(const Compare& comp = Compare(),
                  const Alloc& = Alloc());
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  multimap(Iter first, Iter last,
       const Compare& comp = Compare(), const Alloc& = Alloc());
requires AllocatableElement<Alloc, value_type, const value_type&>
  multimap(const multimap<Key,T,Compare,Alloc>& x);
multimap(multimap<Key,T,Compare,Alloc>&& x);
multimap(const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  multimap(const multimap&, const Alloc&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  multimap(multimap&&, const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  multimap(initializer_list<value_type>,
       const Compare& = Compare(),
       const Allocator& = Allocator());
~multimap();
requires AllocatableElement<Alloc, value_type, const value_type&>
        && CopyAssignable<value_type>
  multimap<Key,T,Compare,Alloc>& operator=(const multimap<Key,T,Compare,Alloc>& x);
multimap<Key,T,Compare,Alloc>&
  operator=(const multimap<Key,T,Compare,Alloc>&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
      && CopyAssignable<value_type>
  multimap<Key,T,Compare,Alloc>& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// iterators:
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;

reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool           empty() const;
size_type      size() const;
size_type      max_size() const;

// modifiers:
```

```
  template <class... Args>
    requires AllocatableElement<Alloc, value_type, Args&&...>
    iterator emplace(Args&&... args);
  template <class... Args>
    requires AllocatableElement<Alloc, value_type, Args&&...>
    iterator emplace_hint(const_iterator position, Args&&... args);
  requires AllocatableElement<Alloc, value_type, const value_type&>}
    iterator insert(const value_type& x);
  template <class P>
    requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
    iterator insert(P&& x);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    iterator insert(const_iterator position, const value_type& x);
  template <class P>
    requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
    iterator insert(const_iterator position, P&& x);
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, value_type, Iter::reference>
          && MoveConstructible<value_type>
    void insert(Iter first, Iter last);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    void insert(initializer_list<value_type>);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(multimap<Key,T,Compare,Alloc>&&);
  void clear();

  // observers:
  key_compare    key_comp() const;
  value_compare  value_comp() const;

  // map operations:
  iterator       find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type      count(const key_type& x) const;

  iterator       lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  iterator       upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;

  pair<iterator,iterator>
    equal_range(const key_type& x);
  pair<const_iterator,const_iterator>
    equal_range(const key_type& x) const;
};

template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
  bool operator==(const multimap<Key,T,Compare,Alloc>& x,
                  const multimap<Key,T,Compare,Alloc>& y);
template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
  bool operator< (const multimap<Key,T,Compare,Alloc>& x,
                  const multimap<Key,T,Compare,Alloc>& y);
```

```
  template <EqualityComparable Key, EqualityComparable T, class Compare, class Alloc>
    bool operator!=(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator> (const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator>=(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);
  template <LessThanComparable Key, LessThanComparable T, class Compare, class Alloc>
    bool operator<=(const multimap<Key,T,Compare,Alloc>& x,
                    const multimap<Key,T,Compare,Alloc>& y);

  // specialized algorithms:
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(multimap<Key,T,Compare,Alloc>& x,
              multimap<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(multimap<Key,T,Compare,Alloc&& x,
              multimap<Key,T,Compare,Alloc>& y);
  template <ValueType Key, ValueType T, class Compare, class Alloc>
    void swap(multimap<Key,T,Compare,Alloc& x,
              multimap<Key,T,Compare,Alloc>&& y);
}
```

23.3.2.1 `multimap` constructors [multimap.cons]

```
explicit multimap(const Compare& comp = Compare(),
                  const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty `multimap` using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  multimap(Iter first, Iter last,
      const Compare& comp = Compare(), const Alloc& = Alloc());
```

3 *Effects:* Constructs an empty `multimap` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

4 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.3.2.2 `multimap` modifiers [multimap.modifiers]

```
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  iterator insert(P&& x);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  iterator insert(const_iterator position, P&& x);
```

1 If P is instantiated as a reference type, then the argument x is copied from. Otherwise x is considered to be an rvalue as it is converted to value_type and inserted into the map. Specifically, in such cases CopyConstructible is not required of key_type or tcodemapped_type unless the conversion from P specifically requires it (e.g. if P is a tuple<const key_type, mapped_type>, then key_type must be CopyConstructible). The signature taking InputIterator parameters does not require CopyConstructible of either key_type or mapped_type if the dereferenced InputIterator returns a non-const rvalue pair<key_type,mapped_type>. Otherwise CopyConstructible is required for both key_type and mapped_type.

23.3.2.3 multimap operations [multimap.ops]

```
iterator       find(const key_type &x);
const_iterator find(const key_type& x) const;

iterator       lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x) const;

pair<iterator, iterator>
  equal_range(const key_type& x);
pair<const_iterator, const_iterator>
  equal_range(const key_type& x) const;
```

1 The find, lower_bound, upper_bound, and equal_range member functions each have two versions, one const and one non-const. In each case the behavior of the two versions is identical except that the const version returns a const_iterator and the non-const version an iterator (23.1.4).

23.3.2.4 multimap specialized algorithms [multimap.special]

```
template <ValueType Key, ValueType T, class Compare, class Alloc>
  void swap(multimap<Key,T,Compare,Alloc>& x,
            multimap<Key,T,Compare,Alloc>& y);
template <ValueType Key, ValueType T, class Compare, class Alloc>
  void swap(multimap<Key,T,Compare,Alloc>&& x,
            multimap<Key,T,Compare,Alloc>& y);
template <ValueType Key, ValueType T, class Compare, class Alloc>
  void swap(multimap<Key,T,Compare,Alloc>& x,
            multimap<Key,T,Compare,Alloc>&& y);
```

1 *Effects:*

```
x.swap(y);
```

23.3.3 Class template set [set]

1 A set is an associative container that supports unique keys (contains at most one of each key value) and provides for fast retrieval of the keys themselves. Class set supports bidirectional iterators.

2 A set satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container (23.1.4), and of an allocator-aware container (Table 82). A set also provides most operations described in (23.1.4) for unique keys. This means that a set supports the a_uniq operations in (23.1.4) but not the a_eq operations. For a set<Key> both the key_type and value_type are Key. Descriptions are provided here only for operations on set that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <ValueType Key, Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<Key> >
  requires NothrowDestructible<Key> && CopyConstructible<Compare>
           && AllocatableElement<Alloc, Compare, const Compare&>
           && AllocatableElement<Alloc, Compare, Compare&&>
  class set {
  public:
    // types:
    typedef Key                                    key_type;
    typedef Key                                    value_type;
    typedef Compare                                key_compare;
    typedef Compare                                value_compare;
    typedef Alloc                                allocator_type;
    typedef typename Alloc::reference            reference;
    typedef typename Alloc::const_reference      const_reference;
    typedef implementation-defined                 iterator;       // See 23.1
    typedef implementation-defined                 const_iterator; // See 23.1
    typedef implementation-defined                 size_type;      // See 23.1
    typedef implementation-defined                 difference_type;// See 23.1
    typedef typename Alloc::pointer              pointer;
    typedef typename Alloc::const_pointer        const_pointer;
    typedef reverse_iterator<iterator>           reverse_iterator;
    typedef reverse_iterator<const_iterator> const_reverse_iterator;

    // 23.3.3.1 construct/copy/destroy:
    explicit set(const Compare& comp = Compare(),
                 const Alloc& = Alloc());
    template <InputIterator Iter>
      requires AllocatableElement<Alloc, value_type, Iter::reference>
            && MoveConstructible<value_type>
      set(Iter first, Iter last,
          const Compare& comp = Compare(), const Alloc& = Alloc());
    requires AllocatableElement<Alloc, value_type, const value_type&>
      set(const set<Key,Compare,Alloc>& x);
    set(set<Key,Compare,Alloc>&& x);
    set(const Alloc&);
    requires AllocatableElement<Alloc, value_type, const value_type&>
      set(const set&, const Alloc&);
    requires AllocatableElement<Alloc, value_type, value_type&&>
      set(set&&, const Alloc&);
    requires AllocatableElement<Alloc, value_type, const value_type&>
      set(initializer_list<value_type>,
          const Compare& = Compare(),
          const Allocator& = Allocator());
   ~set();

    requires AllocatableElement<Alloc, value_type, const value_type&>
          && CopyAssignable<value_type>
      set<Key,Compare,Alloc>& operator=(const set<Key,Compare,Alloc>& x);
    set<Key,Compare,Alloc>& operator=(set<Key,Compare,Alloc>&& x);
    requires AllocatableElement<Alloc, value_type, const value_type&>
          && CopyAssignable<value_type>
      set<Key,Compare,Alloc>& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const;
```

```
// iterators:
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;

reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;

// capacity:
bool          empty() const;
size_type     size() const;
size_type     max_size() const;

// modifiers:
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
  pair<iterator,bool> insert(const value_type& x);
requires AllocatableElement<Alloc, value_type, value_type&&>
  pair<iterator,bool> insert(value_type&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator position, const value_type& x);
requires AllocatableElement<Alloc, value_type, value_type&&>
  iterator insert(const_iterator position, value_type&& x);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference> && MoveConstructible<value_type>
  void insert(Iter first, Iter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
  void insert(initializer_list<value_type>);

iterator  erase(const_iterator position);
size_type erase(const key_type& x);
iterator  erase(const_iterator first, const_iterator last);
void swap(set<Key,Compare,Alloc>&);
void clear();

// observers:
key_compare   key_comp() const;
value_compare value_comp() const;

// set operations:
iterator       find(const key_type& x);
```

```
    const_iterator  find(const key_type& x) const;

    size_type count(const key_type& x) const;

    iterator        lower_bound(const key_type& x);
    const_iterator  lower_bound(const key_type& x) const;

    iterator        upper_bound(const key_type& x);
    const_iterator  upper_bound(const key_type& x) const;

    pair<iterator,iterator>             equal_range(const key_type& x);
    pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
  };

  template <EqualityComparable Key, class Compare, class Alloc>
    bool operator==(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator< (const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <EqualityComparable Key, class Compare, class Alloc>
    bool operator!=(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator> (const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator>=(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);
  template <LessThanComparable Key, class Compare, class Alloc>
    bool operator<=(const set<Key,Compare,Alloc>& x,
                    const set<Key,Compare,Alloc>& y);

  // specialized algorithms:
  template <ValueType Key, class Compare, class Alloc>
    void swap(set<Key,Compare,Alloc>& x,
              set<Key,Compare,Alloc>& y);
  template <ValueType Key, class Compare, class Alloc>
    void swap(set<Key,Compare,Alloc&& x,
              set<Key,Compare,Alloc>& y);
  template <ValueType Key, class Compare, class Alloc>
    void swap(set<Key,Compare,Alloc& x,
              set<Key,Compare,Alloc>&& y);
}
```

23.3.3.1 set constructors, copy, and assignment [set.cons]

```
explicit set(const Compare& comp = Compare(),
             const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty set using the specified comparison objects and allocator.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
```

```
        && MoveConstructible<value_type>
  set(Iter first, Iter last,
      const Compare& comp = Compare(), const Alloc& = Alloc());
```

3 *Effects:* Constructs an empty `set` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

4 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.3.3.2 `set` specialized algorithms [set.special]

```
template <ValueType Key, class Compare, class Alloc>
  void swap(set<Key,Compare,Alloc>& x,
            set<Key,Compare,Alloc>& y);
template <ValueType Key, class Compare, class Alloc>
  void swap(set<Key,Compare,Alloc>&& x,
            set<Key,Compare,Alloc>& y);
template <ValueType Key, class Compare, class Alloc>
  void swap(set<Key,Compare,Alloc>& x,
            set<Key,Compare,Alloc>&& y);
```

1 *Effects:*

```
x.swap(y);
```

23.3.4 Class template `multiset` [multiset]

1 A `multiset` is an associative container that supports equivalent keys (possibly contains multiple copies of the same key value) and provides for fast retrieval of the keys themselves. Class `multiset` supports bidirectional iterators.

2 A `multiset` satisfies all of the requirements of a container, of a reversible container (23.1), of an associative container (23.1.4), and of an allocator-aware container (Table 82). `multiset` also provides most operations described in (23.1.4) for duplicate keys. This means that a `multiset` supports the `a_eq` operations in (23.1.4) but not the `a_uniq` operations. For a `multiset<Key>` both the `key_type` and `value_type` are `Key`. Descriptions are provided here only for operations on `multiset` that are not described in one of these tables and for operations where there is additional semantic information.

```
namespace std {
  template <ValueType Key, Predicate<auto, Key, Key> Compare = less<Key>,
            Allocator Alloc = allocator<Key> >
  requires NothrowDestructible<Key> && CopyConstructible<Compare>
           && AllocatableElement<Alloc, Compare, const Compare&>
           && AllocatableElement<Alloc, Compare, Compare&&>
  class multiset {
  public:
    // types:
    typedef Key                                 key_type;
    typedef Key                                 value_type;
    typedef Compare                             key_compare;
    typedef Compare                             value_compare;
    typedef Alloc                             allocator_type;
    typedef typename Alloc::reference         reference;
    typedef typename Alloc::const_reference   const_reference;
```

```
typedef implementation-defined               iterator;       // See 23.1
typedef implementation-defined               const_iterator; // See 23.1
typedef implementation-defined               size_type;      // See 23.1
typedef implementation-defined               difference_type;// See 23.1
typedef typename Alloc::pointer           pointer;
typedef typename Alloc::const_pointer     const_pointer;
typedef reverse_iterator<iterator>       reverse_iterator;
typedef reverse_iterator<const_iterator> const_reverse_iterator;

// construct/copy/destroy:
explicit multiset(const Compare& comp = Compare(),
                  const Alloc& = Alloc());
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  multiset(Iter first, Iter last,
           const Compare& comp = Compare(),
           const Alloc& = Alloc());
requires AllocatableElement<Alloc, value_type, const value_type&>
  multiset(const multiset<Key,Compare,Alloc>& x);
multiset(multiset<Key,Compare,Alloc>&& x);
multiset(const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  multiset(const multiset&, const Alloc&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  multiset(multiset&&, const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  multiset(initializer_list<value_type>,
       const Compare& = Compare(),
       const Allocator& = Allocator());
~multiset();
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  multiset<Key,Compare,Alloc>& operator=(const multiset<Key,Compare,Alloc>& x);
multiset<Key,Compare,Alloc>& operator=(multiset<Key,Compare,Alloc>&& x);
requires AllocatableElement<Alloc, value_type, const value_type&>
      && CopyAssignable<value_type>
  multiset<Key,Compare,Alloc>& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// iterators:
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;

reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;

const_iterator         cbegin() const;
const_iterator         cend() const;
const_reverse_iterator crbegin() const;
const_reverse_iterator crend() const;
```

```
  // capacity:
  bool          empty() const;
  size_type     size() const;
  size_type     max_size() const;

  // modifiers:
  template <class... Args>
    requires AllocatableElement<Alloc, value_type, Args&&...>
    iterator emplace(Args&&... args);
  template <class... Args>
    requires AllocatableElement<Alloc, value_type, Args&&...>
    iterator emplace_hint(const_iterator position, Args&&... args);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    iterator insert(const value_type& x);
  requires AllocatableElement<Alloc, value_type, value_type&&>
    iterator insert(value_type&& x);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    iterator insert(const_iterator position, const value_type& x);
  requires AllocatableElement<Alloc, value_type, value_type&&>
    iterator insert(const_iterator position, value_type&& x);
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, value_type, Iter::reference> && MoveConstructible<value_type>
    void insert(Iter first, Iter last);
  requires AllocatableElement<Alloc, value_type, const value_type&>
    void insert(initializer_list<value_type>);

  iterator  erase(const_iterator position);
  size_type erase(const key_type& x);
  iterator  erase(const_iterator first, const_iterator last);
  void swap(multiset<Key,Compare,Alloc>&&);
  void clear();

  // observers:
  key_compare   key_comp() const;
  value_compare value_comp() const;

  // set operations:
  iterator        find(const key_type& x);
  const_iterator  find(const key_type& x) const;

  size_type count(const key_type& x) const;

  iterator        lower_bound(const key_type& x);
  const_iterator  lower_bound(const key_type& x) const;

  iterator        upper_bound(const key_type& x);
  const_iterator  upper_bound(const key_type& x) const;

  pair<iterator,iterator>             equal_range(const key_type& x);
  pair<const_iterator,const_iterator> equal_range(const key_type& x) const;
};

template <EqualityComparable Key, class Compare, class Alloc>
  bool operator==(const multiset<Key,Compare,Alloc>& x,
                  const multiset<Key,Compare,Alloc>& y);
```

```
    template <LessThanComparable Key, class Compare, class Alloc>
      bool operator< (const multiset<Key,Compare,Alloc>& x,
                      const multiset<Key,Compare,Alloc>& y);
    template <EqualityComparable Key, class Compare, class Alloc>
      bool operator!=(const multiset<Key,Compare,Alloc>& x,
                      const multiset<Key,Compare,Alloc>& y);
    template <LessThanComparable Key, class Compare, class Alloc>
      bool operator> (const multiset<Key,Compare,Alloc>& x,
                      const multiset<Key,Compare,Alloc>& y);
    template <LessThanComparable Key, class Compare, class Alloc>
      bool operator>=(const multiset<Key,Compare,Alloc>& x,
                      const multiset<Key,Compare,Alloc>& y);
    template <LessThanComparable Key, class Compare, class Alloc>
      bool operator<=(const multiset<Key,Compare,Alloc>& x,
                      const multiset<Key,Compare,Alloc>& y);

    // specialized algorithms:
    template <ValueType Key, class Compare, class Alloc>
      void swap(multiset<Key,Compare,Alloc>& x,
                multiset<Key,Compare,Alloc>& y);
    template <ValueType Key, class Compare, class Alloc>
      void swap(multiset<Key,Compare,Alloc&& x,
                multiset<Key,Compare,Alloc>& y);
    template <ValueType Key, class Compare, class Alloc>
      void swap(multiset<Key,Compare,Alloc& x,
                multiset<Key,Compare,Alloc>&& y);
  }
```

23.3.4.1 `multiset` constructors [multiset.cons]

```
explicit multiset(const Compare& comp = Compare(),
                  const Alloc& = Alloc());
```

1 *Effects:* Constructs an empty set using the specified comparison object and allocator.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
         && MoveConstructible<value_type>
  multiset(Iter first, Iter last,
           const Compare& comp = Compare(),
           const Alloc& = Alloc());
```

3 *Effects:* Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from the range `[first,last)`.

4 *Complexity:* Linear in N if the range `[first,last)` is already sorted using `comp` and otherwise $N \log N$, where N is `last - first`.

23.3.4.2 `multiset` specialized algorithms [multiset.special]

```
template <ValueType Key, class Compare, class Alloc>
  void swap(multiset<Key,Compare,Alloc>& x,
            multiset<Key,Compare,Alloc>& y);
template <ValueType Key, class Compare, class Alloc>
```

```
  void swap(multiset<Key,Compare,Alloc>&& x,
            multiset<Key,Compare,Alloc>& y);
template <ValueType Key, class Compare, class Alloc>
  void swap(multiset<Key,Compare,Alloc>& x,
            multiset<Key,Compare,Alloc>&& y);
```

1 *Effects:*

```
x.swap(y);
```

23.4 Unordered associative containers [unord]

1 Headers <unordered_map> and <unordered_set>:

Header <unordered_map> synopsis

```
namespace std {
  // 23.4.1, class template unordered_map:
  template <ValueType Key,
            ValueType T,
            Callable<auto, const Key&> Hash = hash<Key>,
            Predicate<auto, Key, Key> Pred = equal_to<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
    requires NothrowDestructible<Key> && NothrowDestructible<T>
          && SameType<Hash::result_type, size_t>
          && CopyConstructible<Hash> && CopyConstructible<Pred>
          && AllocatableElement<Alloc, Pred, const Pred&>
          && AllocatableElement<Alloc, Pred, Pred&&>
          && AllocatableElement<Alloc, Hash, const Hash&>
          && AllocatableElement<Alloc, Hash, Hash&&>
    class unordered_map;

  // 23.4.2, class template unordered_multimap:
  template <ValueType Key,
            ValueType T,
            Callable<auto, const Key&> Hash = hash<Key>,
            Predicate<auto, Key, Key> Pred = equal_to<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
    requires NothrowDestructible<Key> && NothrowDestructible<T>
          && SameType<Hash::result_type, size_t>
          && CopyConstructible<Hash> && CopyConstructible<Pred>
          && AllocatableElement<Alloc, Pred, const Pred&>
          && AllocatableElement<Alloc, Pred, Pred&&>
          && AllocatableElement<Alloc, Hash, const Hash&>
          && AllocatableElement<Alloc, Hash, Hash&&>
    class unordered_multimap;

  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>&& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>&& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

```
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>&& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>&& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
} // namespace std
```

Header <unordered_set> synopsis

```
namespace std {
  // 23.4.3, class template unordered_set:
  template <ValueType Value,
            Callable<auto, const Value&> Hash = hash<Value>,
            Predicate<auto, Value, Value> class Pred = equal_to<Value>,
            Allocator Alloc = allocator<Value> >
    requires NothrowDestructible<Value>
          && SameType<Hash::result_type, size_t>
          && CopyConstructible<Hash> && CopyConstructible<Pred>
          && AllocatableElement<Alloc, Pred, const Pred&>
          && AllocatableElement<Alloc, Pred, Pred&&>
          && AllocatableElement<Alloc, Hash, const Hash&>
          && AllocatableElement<Alloc, Hash, Hash&&>
    class unordered_set;

  // 23.4.4, class template unordered_multiset:
  template <ValueType Value,
            Callable<auto, const Value&> Hash = hash<Value>,
            Predicate<auto, Value, Value> class Pred = equal_to<Value>,
            Allocator Alloc = allocator<Value> >
    requires NothrowDestructible<Value>
          && SameType<Hash::result_type, size_t>
          && CopyConstructible<Hash> && CopyConstructible<Pred>
          && AllocatableElement<Alloc, Pred, const Pred&>
          && AllocatableElement<Alloc, Pred, Pred&&>
          && AllocatableElement<Alloc, Hash, const Hash&>
          && AllocatableElement<Alloc, Hash, Hash&&>
    class unordered_multiset;

  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>&& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>&& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);

  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
              unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

```
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
              unordered_multiset<Value, Hash, Pred, Alloc>&& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>&& x,
              unordered_multiset<Value, Hash, Pred, Alloc>& y);
} // namespace std
```

23.4.1 Class template `unordered_map` [unord.map]

1 An `unordered_map` is an unordered associative container that supports unique keys (an `unordered_map` contains at most one of each key value) and that associates values of another type `mapped_type` with the keys.

2 An `unordered_map` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 82). It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_map` supports the `a_uniq` operations in that table, not the `a_eq` operations. For an `unordered_map<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `pair<const Key, T>`.

3 This section only describes operations on `unordered_map` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <ValueType Key,
            ValueType T,
            Callable<auto, const Key&> Hash = hash<Key>,
            Predicate<auto, Key, Key> Pred = equal_to<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
  requires NothrowDestructible<Key> && NothrowDestructible<T>
        && SameType<Hash::result_type, size_t>
        && CopyConstructible<Hash> && CopyConstructible<Pred>
        && AllocatableElement<Alloc, Pred, const Pred&>
        && AllocatableElement<Alloc, Pred, Pred&&>
        && AllocatableElement<Alloc, Hash, const Hash&>
        && AllocatableElement<Alloc, Hash, Hash&&>
  class unordered_map
  {
  public:
    // types
    typedef Key                                        key_type;
    typedef pair<const Key, T>                   value_type;
    typedef T                                          mapped_type;
    typedef Hash                                       hasher;
    typedef Pred                                       key_equal;
    typedef Alloc                                      allocator_type;
    typedef typename allocator_type::pointer           pointer;
    typedef typename allocator_type::const_pointer     const_pointer;
    typedef typename allocator_type::reference         reference;
    typedef typename allocator_type::const_reference   const_reference;
    typedef implementation-defined                     size_type;
    typedef implementation-defined                     difference_type;

    typedef implementation-defined                     iterator;
    typedef implementation-defined                     const_iterator;
    typedef implementation-defined                     local_iterator;
```

```
typedef implementation-defined                  const_local_iterator;

// construct/destroy/copy
explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  unordered_map(Iter f, Iter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
requires AllocatableElement<Alloc, value_type, const value_type&>
  unordered_map(const unordered_map&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  unordered_map(unordered_map&&);
unordered_map(const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  unordered_map(const unordered_map&, const Alloc&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  unordered_map(unordered_map&&, const Alloc&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
   unordered_map(initializer_list<value_type>,
                 size_type = implementation-defined,
                 const hasher& hf = hasher(),
                 const key_equal& eql = key_equal(),
                 const allocator_type& a = allocator_type());
~unordered_map();
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_map& operator=(const unordered_map&);
requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
  unordered_map& operator=(unordered_map&&);
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_map& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
```

```
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
  pair<iterator, bool> insert(const value_type& obj);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  pair<iterator, bool> insert(P&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  pair<iterator, bool> insert(const_iterator hint, P&& obj);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  void insert(Iter first, Iter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
  void insert(initializer_list<value_type>);
iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_map&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator       find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator>             equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

requires AllocatableElement<Alloc, value_type, const key_type&, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  mapped_type& operator[](const key_type& k);
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  mapped_type& operator[](key_type&& k);
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
```

```
    const_local_iterator end(size_type n) const;
    const_local_iterator cbegin(size_type n) const;
    const_local_iterator cend(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    requires MoveConstructible<value_type> void rehash(size_type n);
  };

  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
              unordered_map<Key, T, Hash, Pred, Alloc>&& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_map<Key, T, Hash, Pred, Alloc>&& x,
              unordered_map<Key, T, Hash, Pred, Alloc>& y);
}
```

23.4.1.1 `unordered_map` constructors [unord.map.cnstr]

```
explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
         && MoveConstructible<value_type>
  unordered_map(Iter f, Iter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[f, l)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.1.2 `unordered_map` element access [unord.map.elem]

```
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
     && AllocatableElement<Alloc, mapped_type>
  mapped_type& operator[](const key_type& k);
```

```
requires AllocatableElement<Alloc, value_type, key_type&&, mapped_type&&>
      && AllocatableElement<Alloc, mapped_type>
  mapped_type& operator[](key_type&& k);
```

1 *Effects:* If the `unordered_map` does not already contain an element whose key is equivalent to ***k***, inserts the value `pair<const key_type, mapped_type>(k, mapped_type())` or `pair<const key_type, mapped_type>(move(k), mapped_type())`, respectively.

2 *Returns:* A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to ***k***.

```
mapped_type& at(const key_type& k);
const mapped_type& at(const key_type& k) const;
```

3 *Returns:* A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to `k`.

4 *Throws:* An exception object of type `out_of_range` if no such element is present.

23.4.1.3 `unordered_map` swap [unord.map.swap]

```
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
  void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
            unordered_map<Key, T, Hash, Pred, Alloc>& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
  void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
            unordered_map<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
  void swap(unordered_map<Key, T, Hash, Pred, Alloc>&& x,
            unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.4.2 Class template `unordered_multimap` [unord.multimap]

1 An `unordered_multimap` is an unordered associative container that supports equivalent keys (an `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys.

2 An `unordered_multimap` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 82). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `pair<const Key, T>`.

3 This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <ValueType Key,
            ValueType T,
            Callable<auto, const Key&> Hash = hash<Key>,
            Predicate<auto, Key, Key> Pred = equal_to<Key>,
            Allocator Alloc = allocator<pair<const Key, T> > >
  requires NothrowDestructible<Key> && NothrowDestructible<T>
        && SameType<Hash::result_type, size_t>
        && CopyConstructible<Hash> && CopyConstructible<Pred>
        && AllocatableElement<Alloc, Pred, const Pred&>
```

```
      && AllocatableElement<Alloc, Pred, Pred&&>
      && AllocatableElement<Alloc, Hash, const Hash&>
      && AllocatableElement<Alloc, Hash, Hash&&>
class unordered_multimap
{
public:
  // types
  typedef Key                                        key_type;
  typedef pair<const Key, T>                  value_type;
  typedef T                                          mapped_type;
  typedef Hash                                       hasher;
  typedef Pred                                       key_equal;
  typedef Alloc                                      allocator_type;
  typedef typename allocator_type::pointer           pointer;
  typedef typename allocator_type::const_pointer     const_pointer;
  typedef typename allocator_type::reference         reference;
  typedef typename allocator_type::const_reference const_reference;
  typedef implementation-defined                     size_type;
  typedef implementation-defined                     difference_type;

  typedef implementation-defined                     iterator;
  typedef implementation-defined                     const_iterator;
  typedef implementation-defined                     local_iterator;
  typedef implementation-defined                     const_local_iterator;

  // construct/destroy/copy
  explicit unordered_multimap(size_type n = implementation-defined,
                              const hasher& hf = hasher(),
                              const key_equal& eql = key_equal(),
                              const allocator_type& a = allocator_type());
  template <InputIterator Iter>
    requires AllocatableElement<Alloc, value_type, Iter::reference>
          && MoveConstructible<value_type>
    unordered_multimap(Iter f, Iter l,
                       size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
  requires AllocatableElement<Alloc, value_type, const value_type&>
    unordered_multimap(const unordered_multimap&);
  requires AllocatableElement<Alloc, value_type, value_type&&>
    unordered_multimap(unordered_multimap&&);
  unordered_multimap(const Alloc&);
  requires AllocatableElement<Alloc, value_type, const value_type&>
     unordered_multimap(const unordered_multimap&, const Alloc&);
  requires AllocatableElement<Alloc, value_type, value_type&&>
     unordered_multimap(unordered_multimap&&, const Alloc&);
   requires AllocatableElement<Alloc, value_type, const value_type&>
     unordered_multimap(initializer_list<value_type>,
                        size_type = implementation-defined,
                        const hasher& hf = hasher(),
                        const key_equal& eql = key_equal(),
                        const allocator_type& a = allocator_type());
  ~unordered_multimap();
  requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
```

```
  unordered_multimap& operator=(const unordered_multimap&);
requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
  unordered_multimap& operator=(unordered_multimap&&);
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_multimap& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const value_type& obj);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  iterator insert(P&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
template <class P>
  requires AllocatableElement<Alloc, value_type, P&&> && MoveConstructible<value_type>
  iterator insert(const_iterator hint, P&& obj);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  void insert(Iter first, Iter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
  void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multimap&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;
```

```
    // lookup
    iterator       find(const key_type& k);
    const_iterator find(const key_type& k) const;
    size_type count(const key_type& k) const;
    pair<iterator, iterator>             equal_range(const key_type& k);
    pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

    // bucket interface
    size_type bucket_count() const;
    size_type max_bucket_count() const;
    size_type bucket_size(size_type n);
    size_type bucket(const key_type& k) const;
    local_iterator begin(size_type n);
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;
    const_local_iterator cbegin(size_type n) const;
    const_local_iterator cend(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    requires MoveConstructible<value_type> void rehash(size_type n);
  };

  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>&& y);
  template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>&& x,
              unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
}
```

23.4.2.1 `unordered_multimap` constructors [unord.multimap.cnstr]

```
explicit unordered_multimap(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  unordered_multimap(Iter f, Iter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
```

```
    const key_equal& eql = key_equal(),
    const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[`*f*`,` *l*`)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.2.2 `unordered_multimap` swap [unord.multimap.swap]

```
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
  void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
            unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
  void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
            unordered_multimap<Key, T, Hash, Pred, Alloc>&& y);
template <ValueType Key, ValueType T, class Hash, class Pred, class Alloc>
  void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>&& x,
            unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.4.3 Class template `unordered_set` [unord.set]

1 An `unordered_set` is an unordered associative container that supports unique keys (an `unordered_set` contains at most one of each key value) and in which the elements' keys are the elements themselves.

2 An `unordered_set` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 82). It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_set` supports the `a_uniq` operations in that table, not the `a_eq` operations. For an `unordered_set<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether they are the same type.

3 This section only describes operations on `unordered_set` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <ValueType Value,
            Callable<auto, const Value&> Hash = hash<Value>,
            Predicate<auto, Value, Value> class Pred = equal_to<Value>,
            Allocator Alloc = allocator<Value> >
  requires NothrowDestructible<Value>
        && SameType<Hash::result_type, size_t>
        && CopyConstructible<Hash> && CopyConstructible<Pred>
        && AllocatableElement<Alloc, Pred, const Pred&>
        && AllocatableElement<Alloc, Pred, Pred&&>
        && AllocatableElement<Alloc, Hash, const Hash&>
        && AllocatableElement<Alloc, Hash, Hash&&>
  class unordered_set
  {
  public:
    // types
    typedef Value                                    key_type;
    typedef Value                                    value_type;
```

```
typedef Hash                                    hasher;
typedef Pred                                    key_equal;
typedef Alloc                                   allocator_type;
typedef typename allocator_type::pointer        pointer;
typedef typename allocator_type::const_pointer  const_pointer;
typedef typename allocator_type::reference      reference;
typedef typename allocator_type::const_reference const_reference;
typedef implementation-defined                  size_type;
typedef implementation-defined                  difference_type;

typedef implementation-defined                  iterator;
typedef implementation-defined                  const_iterator;
typedef implementation-defined                  local_iterator;
typedef implementation-defined                  const_local_iterator;

// construct/destroy/copy
explicit unordered_set(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  unordered_set(Iter f, Iter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
requires AllocatableElement<Alloc, value_type, const value_type&>
  unordered_set(const unordered_set&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  unordered_set(unordered_set&&);
unordered_set(const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  unordered_set(const unordered_set&, const Alloc&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  unordered_set(unordered_set&&, const Alloc&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
   unordered_set(initializer_list<value_type>,
                 size_type = implementation-defined,
                 const hasher& hf = hasher(),
                 const key_equal& eql = key_equal(),
                 const allocator_type& a = allocator_type());
~unordered_set();
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_set& operator=(const unordered_set&);
requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
  unordered_set& operator=(unordered_set&&);
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_set& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
```

```
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;

// modifiers
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  pair<iterator, bool> emplace(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
  pair<iterator, bool> insert(const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
  pair<iterator, bool> insert(value_type&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
  iterator insert(const_iterator hint, value_type&& obj);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  void insert(Iter first, Iter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
  void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_set&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator       find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator>             equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
```

```
    local_iterator begin(size_type n);
    const_local_iterator begin(size_type n) const;
    local_iterator end(size_type n);
    const_local_iterator end(size_type n) const;
    const_local_iterator cbegin(size_type n) const;
    const_local_iterator cend(size_type n) const;

    // hash policy
    float load_factor() const;
    float max_load_factor() const;
    void max_load_factor(float z);
    requires MoveConstructible<value_type> void rehash(size_type n);
  };

  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>&& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>&& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);
}
```

23.4.3.1 `unordered_set` constructors [unord.set.cnstr]

```
explicit unordered_set(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  unordered_set(Iter f, Iter l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[f, l)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.3.2 `unordered_set` swap [unord.set.swap]

```
template <ValueType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
            unordered_set<Value, Hash, Pred, Alloc>& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
            unordered_set<Value, Hash, Pred, Alloc>&& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_set<Value, Hash, Pred, Alloc>&& x,
            unordered_set<Value, Hash, Pred, Alloc>& y);
```

1 *Effects:* `x.swap(y)`.

23.4.4 Class template `unordered_multiset` [unord.multiset]

1 An `unordered_multiset` is an unordered associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which each element's key is the element itself.

2 An `unordered_multiset` satisfies all of the requirements of a container, of an unordered associative container, and of an allocator-aware container (Table 82). It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multiset<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether they are the same type.

3 This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
namespace std {
  template <ValueType Value,
            Callable<auto, const Value&> Hash = hash<Value>,
            Predicate<auto, Value, Value> class Pred = equal_to<Value>,
            Allocator Alloc = allocator<Value> >
  requires NothrowDestructible<Value>
        && SameType<Hash::result_type, size_t>
        && CopyConstructible<Hash> && CopyConstructible<Pred>
        && AllocatableElement<Alloc, Pred, const Pred&>
        && AllocatableElement<Alloc, Pred, Pred&&>
        && AllocatableElement<Alloc, Hash, const Hash&>
        && AllocatableElement<Alloc, Hash, Hash&&>
  class unordered_multiset
  {
  public:
    // types
    typedef Value                                    key_type;
    typedef Value                                    value_type;
    typedef Hash                                     hasher;
    typedef Pred                                     key_equal;
    typedef Alloc                                    allocator_type;
    typedef typename allocator_type::pointer         pointer;
    typedef typename allocator_type::const_pointer   const_pointer;
    typedef typename allocator_type::reference       reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined                   size_type;
    typedef implementation-defined                   difference_type;
```

```
typedef implementation-defined                  iterator;
typedef implementation-defined                  const_iterator;
typedef implementation-defined                  local_iterator;
typedef implementation-defined                  const_local_iterator;

// construct/destroy/copy
explicit unordered_multiset(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  unordered_multiset(Iter f, Iter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
requires AllocatableElement<Alloc, value_type, const value_type&>
  unordered_multiset(const unordered_multiset&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  unordered_multiset(unordered_multiset&&);
unordered_multiset(const Alloc&);
requires AllocatableElement<Alloc, value_type, const value_type&>
  unordered_multiset(const unordered_multiset&, const Alloc&);
requires AllocatableElement<Alloc, value_type, value_type&&>
  unordered_multiset(unordered_multiset&&, const Alloc&);
 requires AllocatableElement<Alloc, value_type, const value_type&>
   unordered_multiset(initializer_list<value_type>,
                      size_type = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
~unordered_multiset();
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_multiset& operator=(const unordered_multiset&);
requires AllocatableElement<Alloc, value_type, value_type&&> && MoveAssignable<value_type>
  unordered_multiset& operator=(unordered_multiset&&);
requires AllocatableElement<Alloc, value_type, const value_type&> && CopyAssignable<value_type>
  unordered_multiset& operator=(initializer_list<value_type>);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
const_iterator cbegin() const;
const_iterator cend() const;
```

```
// modifiers
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace(Args&&... args);
template <class... Args>
  requires AllocatableElement<Alloc, value_type, Args&&...>
  iterator emplace_hint(const_iterator position, Args&&... args);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
  iterator insert(value_type&& obj);
requires AllocatableElement<Alloc, value_type, const value_type&>
  iterator insert(const_iterator hint, const value_type& obj);
requires AllocatableElement<Alloc, value_type, value_type&&>
  iterator insert(const_iterator hint, value_type&& obj);
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::value_type>
        && MoveConstructible<value_type>
  void insert(Iter first, Iter last);
requires AllocatableElement<Alloc, value_type, const value_type&>
  void insert(initializer_list<value_type>);

iterator erase(const_iterator position);
size_type erase(const key_type& k);
iterator erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multiset&&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator       find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
pair<iterator, iterator>             equal_range(const key_type& k);
pair<const_iterator, const_iterator> equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;
const_local_iterator cbegin(size_type n) const;
const_local_iterator cend(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
```

```
    void max_load_factor(float z);
    requires MoveConstructible<value_type> void rehash(size_type n);
  };

  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
              unordered_multiset<Value, Hash, Pred, Alloc>& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
              unordered_multiset<Value, Hash, Pred, Alloc>&& y);
  template <ValueType Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multiset<Value, Hash, Pred, Alloc>&& x,
              unordered_multiset<Value, Hash, Pred, Alloc>& y);
}
```

23.4.4.1 `unordered_multiset` constructors [unord.multiset.cnstr]

```
explicit unordered_multiset(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

1 *Effects:* Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` returns 1.0.

2 *Complexity:* Constant.

```
template <InputIterator Iter>
  requires AllocatableElement<Alloc, value_type, Iter::reference>
        && MoveConstructible<value_type>
  unordered_multiset(Iter f, Iter l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

3 *Effects:* Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[f, l)`. `max_load_factor()` returns 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

23.4.4.2 `unordered_multiset` swap [unord.multiset.swap]

```
template <ValueType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
            unordered_multiset<Value, Hash, Pred, Alloc>& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
            unordered_multiset<Value, Hash, Pred, Alloc>&& y);
template <ValueType Value, class Hash, class Pred, class Alloc>
  void swap(unordered_multiset<Value, Hash, Pred, Alloc>&& x,
            unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y);`

24 Iterators library [iterators]

1 This Clause describes components that C++ programs may use to perform iterations over containers (Clause 23), streams (27.6), and stream buffers (27.5).

2 The following subclauses describe iterator concepts, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 88.

Table 88 — Iterators library summary

	Subclause	Header(s)
24.1	Concepts	`<iterator_concepts>`
24.3	Iterator operations	`<iterator>`
24.4	Predefined iterators	
24.5	Stream iterators	
D.10	Iterator primitives	

24.1 Iterator concepts [iterator.concepts]

1 The `<iterator_concepts>` header describes requirements on iterators.

Header `<iterator_concepts>` synopsis

```
namespace std {
  concept Iterator<typename X> see below;

  // 24.1.2, input iterators:
  concept InputIterator<typename X> see below;

  // 24.1.3, output iterators:
  auto concept OutputIterator<typename X, typename Value> see below;

  // 24.1.4, forward iterators:
  concept ForwardIterator<typename X> see below;

  // 24.1.5, bidirectional iterators:
  concept BidirectionalIterator<typename X> see below;

  // 24.1.6, random access iterators:
  concept RandomAccessIterator<typename X> see below;
  template<ObjectType T> concept_map RandomAccessIterator<T*> see below;
  template<ObjectType T> concept_map RandomAccessIterator<const T*> see below;

  // 24.1.7, shuffle iterators:
  auto concept ShuffleIterator<typename X> see below;

  // 24.1.8, ranges:
  concept Range<typename T> see below;

  template<class T, size_t N>
```

```
    concept_map Range<T[N]> see below;
}
```

2 Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. All iterators meet the requirements of the `Iterator` concept. All input iterators `i` support the expression `*i`, resulting in a value of some class, enumeration, or built-in type `T`, called the *value type* of the iterator. All output iterators support the expression `*i = o` where `o` is a value of some type that is in the set of types that are *writable* to the particular iterator type of `i`. All iterators `i` for which the expression `(*i).m` is well-defined, support the expression `i->m` with the same semantics as `(*i).m`. For every iterator type `X` for which equality is defined, there is a corresponding signed integral type called the *difference type* of the iterator.

3 Since iterators are an abstraction of pointers, their semantics are a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This International Standard defines several iterator concepts, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators*, *random access iterators*, and *shuffle iterators*, as shown in Table 89.

Table 89 — Relations among iterator categories

Random Access	→	**Bidirectional**	→	**Forward**	→	**Input**	→	**Iterator**
				↑				↑
				Shuffle			→	**Output**

4 Forward iterators satisfy all the requirements of the input iterators and can be used whenever an input iterator is specified; Bidirectional iterators also satisfy all the requirements of the forward iterators and can be used whenever a forward iterator is specified; Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

5 Iterators that meet the requirements of the `OutputIterator` concept are called *mutable iterators*. Non-mutable iterators are referred to as *constant iterators*.

6 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any container. [*Example:* After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a singular value of a pointer. —*end example*] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value and the assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

7 An iterator `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == j`. If `j` is reachable from `i`, they refer to the same container.

8 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation. A range `[i,i)` is an empty range; in general, a range `[i,j)` refers to the elements in the data structure starting with the one pointed to by `i` and up to but not including the one pointed to by `j`. Range `[i,j)` is valid if and only if `j` is reachable from `i`. The result of the application of functions in the library to invalid ranges is undefined.

9 All the iterator concepts require only those functions that are realizable in constant time (amortized).

10 Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

11 An *invalid iterator* is an iterator that may be singular.[262]

24.1.1 Iterator [iterator.iterators]

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;

  requires HasDereference<postincrement_result>;

  reference operator*(X&&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
}
```

1 The `Iterator` concept forms the basis of the iterator concept taxonomy, and every iterator meets the requirements of the `Iterator` concept. This concept specifies operations for dereferencing and incrementing the iterator, but provides no way to manipulate values. Most algorithms will require additional operations to read (24.1.2) or write (24.1.3) values, or to provide a richer set of iterator movements (24.1.4, 24.1.5, 24.1.6).

```
reference operator*(X&& a);
```

2 *Requires:* `a` is dereferenceable.

```
postincrement_result operator++(X& r, int);
```

3 *Effects:* equivalent to `{ X tmp = r; ++r; return tmp; }`.

24.1.2 Input iterators [input.iterators]

1 A class or a built-in type `X` satisfies the requirements of an input iterator for the value type `T` if it meets the syntactic and semantic requirements of the `InputIterator` concept.

```
concept InputIterator<typename X> : Iterator<X>, EqualityComparable<X> {
  ObjectType value_type = typename X::value_type;
  MoveConstructible pointer = typename X::pointer;

  SignedIntegralLike difference_type = typename X::difference_type;

  requires IntegralType<difference_type>
        && Convertible<reference, const value_type &>;
        && Convertible<pointer, const value_type*>;

  requires Convertible<HasDereference<postincrement_result>::result_type, const value_type&>;

  pointer operator->(const X&);
}
```

262) This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

2 In the `InputIterator` concept, the term *the domain of* `==` is used in the ordinary mathematical sense to denote the set of values over which `==` is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of `==` for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of `==` and `!=`. [*Example:* the call `find(a,b,x)` is defined only if the value of `a` has the property p defined as follows: `b` has property p and a value `i` has property p if `(*i==x)` or if `(*i!=x` and `++i` has property `p`). — *end example*]

3 [*Note:* For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class. — *end note*]

`reference operator*(X&& a);` *// inherited from* `Iterator<X>`

4 *Returns:* the value referenced by the iterator

5 *Remarks:* If `b` is a value of type `X`, `a == b` and `(a, b)` is in the domain of `==` then `*a` is equivalent to `*b`.

`pointer operator->(const X& a);`

6 *Returns:* a pointer to the value referenced by the iterator

`bool operator==(const X& a, const X& b);` *// inherited from* `EqualityComparable<X>`

7 If two iterators `a` and `b` of the same type are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

`X& operator++(X& r);`

8 *Precondition:* `r` is dereferenceable

9 *Postcondition:* `r` is dereferenceable or `r` is past-the-end. Any copies of the previous value of `r` are no longer required either to be dereferenceable or in the domain of `==`.

24.1.3 Output iterators [output.iterators]

1 A class or a built-in type `X` satisfies the requirements of an output iterator meets the syntactic and semantic requirements of the `OutputIterator` concept.

2 [*Note:* The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — *end note*]

3 The `OutputIterator` concept describes an output iterator that may permit output of many different value types.

```
auto concept OutputIterator<typename X, typename Value> {
  requires Iterator<X>;

  typename reference = Iterator<X>::reference;
  typename postincrement_result = Iterator<X>::postincrement_result;
  requires SameType<reference, Iterator<X>::reference>
        && SameType<postincrement_result, Iterator<X>::postincrement_result>
        && Convertible<postincrement_result, const X&>
```

```
        && HasAssign<reference, Value>
        && HasAssign<HasDereference<postincrement_result>::result_type, Value>;
}
```

4 [*Note:* Any iterator that meets the additional requirements specified by `OutputIterator` for a given `Value` type is considered an output iterator. — *end note*]

```
X& operator++(X& r); // from Iterator<X>
```

5 *Postcondition:* `&r == &++r`

24.1.4 Forward iterators [forward.iterators]

1 A class or a built-in type `X` satisfies the requirements of a forward iterator if it meets the syntactic and semantic requirements of the `ForwardIterator` concept.

```
concept ForwardIterator<typename X> : InputIterator<X>, Regular<X> {
  requires Convertible<postincrement_result, const X&>;

  axiom MultiPass(X a, X b) {
    if (a == b) *a == *b;
    if (a == b) ++a == ++b;
  }
}
```

2 [*Note:* The axiom that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

```
X::X(); // inherited from Regular<X>
```

3 *Note:* the constructed object might have a singular value.

```
X& operator++(X& r); // inherited from InputIterator<X>
```

4 *Postcondition:* `&r == &++r`.

24.1.5 Bidirectional iterators [bidirectional.iterators]

1 A class or a built-in type `X` satisfies the requirements of a bidirectional iterator if it meets the syntactic and semantic requirements of the `BidirectionalIterator` concept.

```
concept BidirectionalIterator<typename X> : ForwardIterator<X> {
  MoveConstructible postdecrement_result;
  requires HasDereference<postdecrement_result>
        &&   Convertible<HasDereference<postdecrement_result>::result_type, const value_type&>
        &&   Convertible<postdecrement_result, const X&>;

  X& operator--(X&);
  postdecrement_result operator--(X&, int);
}
```

2 [*Note:* Bidirectional iterators allow algorithms to move iterators backward as well as forward. — *end note*]

```
X& operator--(X& r);
```

3 *Precondition:* there exists `s` such that `r == ++s`.

4 *Requires:* `--(++r) == r` and, given lvalues `a` and `b` of type `X`, `--a == --b` implies `a == b`

5 *Postcondition:* `r` is dereferenceable. `&r == &--r`.

```
postdecrement_result operator--(X& r, int);
```

6 *Effects:* equivalent to

```
{ X tmp = r;
--r;
return tmp; }
```

24.1.6 Random access iterators [random.access.iterators]

1 A class or a built-in type `X` satisfies the requirements of a random access iterator if it meets the syntactic and semantic requirements of the `RandomAccessIterator` concept.

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  MoveConstructible subscript_reference;
  requires Convertible<subscript_reference, const value_type&>;

  X& operator+=(X&, difference_type);
  X  operator+ (const X& x, difference_type n) { X tmp(x); tmp += n; return tmp; }
  X  operator+ (difference_type n, const X& x) { X tmp(x); tmp += n; return tmp; }
  X& operator-=(X&, difference_type);
  X  operator- (const X& x, difference_type n) { X tmp(x); tmp -= n; return tmp; }

  difference_type operator-(const X&, const X&);
  subscript_reference operator[](const X& x, difference_type n);
}
```

```
X& operator+=(X& r, difference_type n);
```

2 *Effects:* equivalent to

```
{ difference_type m = n;
  if (m >= 0) while (m--) ++r;
  else while (m++) --r;
  return r; }
```

```
X operator+(const X& a, difference_type n);
X operator+(difference_type n, const X& a);
```

3 *Effects:* equivalent to

```
{ X tmp = a;
return tmp += n; }
```

4 *Postcondition:* `a + n == n + a`

```
X& operator-=(X& r, difference_type n);
```

5 *Returns:* `r += -n`

```
X operator-(const X& a, difference_type n);
```

6 *Effects:* equivalent to

```
{ X tmp = a;
  return tmp -= n; }
```

```
difference_type operator-(const X& a, const X& b);
```

7 *Precondition:* there exists a value `n` of `difference_type` such that `a == b + n`.

8 *Effects:* `b == a + (b - a)`

9 *Returns:* `(a < b) ? distance(a,b) : -distance(b,a)`

```
subscript_reference operator[](const X& x, difference_type n);
```

10 *Requires:* `(const value_type&)x[n]` is equivalent to `*(x + n)`.

11 Pointers are random access iterators with the following concept map

```
namespace std {
  template<ObjectType T> concept_map RandomAccessIterator<T*> {
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef T& reference;
    typedef T* pointer;
  }
}
```

and pointers to const are random access iterators

```
namespace std {
  template<ObjectType T> concept_map RandomAccessIterator<const T*> {
    typedef T value_type;
    typedef ptrdiff_t difference_type;
    typedef const T& reference;
    typedef const T* pointer;
  }
}
```

12 [*Note:* If there is an additional pointer type `__far` such that the difference of two `__far` pointers is of type `long`, an implementation may define

```
template <ObjectType T> concept_map RandomAccessIterator<T __far*> {
  typedef long difference_type;
  typedef T value_type;
  typedef T __far* pointer;
  typedef T __far& reference;
}

template <ObjectType T> concept_map RandomAccessIterator<const T __far*> {
  typedef long difference_type;
  typedef T value_type;
  typedef const T __far* pointer;
  typedef const T __far& reference;
}
```

— *end note*]

24.1.7 Shuffle iterators [shuffle.iterators]

1 A class or built-in type `X` satisfies the requirements of a shuffle iterator if it meets the syntactic and semantic requirements of the `ShuffleIterator` concept.

```
auto concept ShuffleIterator<typename X> {
  requires ForwardIterator<X>
        && OutputIterator<X, RvalueOf<ForwardIterator<X>::value_type>::type>
        && OutputIterator<X, RvalueOf<ForwardIterator<X>::reference>::type>
        && Constructible<ForwardIterator<X>::value_type,
                         RvalueOf<ForwardIterator<X>::reference>::type>
        && MoveConstructible<ForwardIterator<X>::value_type>
        && MoveAssignable<ForwardIterator<X>::value_type>
        && Swappable<ForwardIterator<X>::value_type>
        && HasAssign<ForwardIterator<X>::value_type,
                     RvalueOf<ForwardIterator<X>::reference>::type>
        && HasSwap<ForwardIterator<X>::reference, ForwardIterator<X>::reference>;
}
```

2 A shuffle iterator is a form of forward and output iterator that allows values to be moved into or out of a sequence, along with permitting efficient swapping of values within the sequence. Shuffle iterators are typically used in algorithms that need to rearrange the elements within a sequence in a way that cannot be performed efficiently with swaps alone.

3 [*Note:* Any iterator that meets the additional requirements specified by `ShuffleIterator` is considered a shuffle iterator. — *end note*]

24.1.8 Ranges [iterator.concepts.range]

1 A type `T` satisfies the requirements of a range if it meets the syntactic and semantic requirements of the `Range` concept.

```
concept Range<typename T> {
  InputIterator iterator;
  iterator begin(T&);
  iterator end(T&);
}
```

2 *Note:* any object of a type meeting the requirements of the `Range` concept can be used with the range-based `for` statement (6.5.4).

3 *Requires:* for an object `t` of a type `T` that meets the requirements of the `Range` concept, `[Range<T>::begin(t), Range<T>::end(t))` shall designate a valid range (24.1).

```
template<class T, size_t N>
concept_map Range<T[N]> {
  typedef T* iterator;
  iterator begin(T(&a)[N]) { return a; }
  iterator end(T(&a)[N]) { return a + N; }
}
```

4 *Note:* adapts an array to the `Range` concept.

24.2 Header `<iterator>` synopsis [iterator.synopsis]

```
namespace std {
  // D.10, primitives:
```

```
template<class Iterator> struct iterator_traits;
template<class T> struct iterator_traits<T*>;

template<class Category, class T, class Distance = ptrdiff_t,
     class Pointer = T*, class Reference = T&> struct iterator;

struct input_iterator_tag { };
struct output_iterator_tag { };
struct forward_iterator_tag: public input_iterator_tag { };
struct bidirectional_iterator_tag: public forward_iterator_tag { };
struct random_access_iterator_tag: public bidirectional_iterator_tag { };

// 24.3, iterator operations:
template <InputIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
template <BidirectionalIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
template <RandomAccessIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
template <InputIterator Iter>
  Iter::difference_type
  distance(Iter first, Iter last); % no line break before distance
template <RandomAccessIterator Iter>
  Iter::difference_type
  distance(Iter first, Iter last);
template <InputIterator Iter>
  Iter next(Iter x,
    Iter::difference_type n = 1);
template <BidirectionalIterator Iter>
  Iter prev(Iter x,
    Iter::difference_type n = 1);

// 24.4, predefined iterators:
template <BidirectionalIterator Iter> class reverse_iterator;

template <BidirectionalIterator Iter1, BidirectionalIterator Iter2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasGreater<Iter1, Iter2>
  bool operator<(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
template <BidirectionalIterator Iter1, BidirectionalIterator Iter2>
  requires HasNotEqualTo<Iter1, Iter2>
  bool operator!=(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter1, Iter2>
  bool operator>(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLessEqual<Iter1, Iter2>
  bool operator>=(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasGreaterEqual<Iter1, Iter2>
  bool operator<=(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasMinus<Iter2, Iter1>
  auto operator-(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y) -> decltype(y.base() - x.base());
template <RandomAccessIterator Iterator>
  reverse_iterator<Iter> operator+(
    Iter::difference_type n,
    const reverse_iterator<Iter>& x);

template<BidirectionalIterator Iter>
concept_map BidirectionalIterator<reverse_iterator<Iter> > { }

template<RandomAccessIterator Iter>
concept_map RandomAccessIterator<reverse_iterator<Iter> > { }

template <BackInsertionContainer Cont> class back_insert_iterator;
template <BackInsertionContainer Cont>
  back_insert_iterator<Cont> back_inserter(Cont& x);
template<BackInsertionContainer Cont>
  concept_map Iterator<back_insert_iterator<Cont> > { }

template <FrontInsertionContainer Cont> class front_insert_iterator;
template <FrontInsertionContainer Cont>
  front_insert_iterator<Cont> front_inserter(Cont& x);
template<FrontInsertionContainer Cont>
  concept_map Iterator<front_insert_iterator<Cont> > { }

template <InsertionContainer Cont> class insert_iterator;
template <InsertionContainer Cont>
  insert_iterator<Cont> inserter(Cont& x, Cont::iterator i);
template<InsertionContainer Cont>
  concept_map Iterator<insert_iterator<Cont> > { }

template <InputIterator Iter> class move_iterator;
template <InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
template <InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator!=(
    const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter1, Iter2>
```

```
  bool operator<(
    const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter2, Iter1>
  bool operator<=(
    const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter2, Iter1>
  bool operator>(
    const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter1, Iter2>
  bool operator>=(
    const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);

template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasMinus<Iter1, Iter2>
  auto operator-(
    const move_iterator<Iter1>& x,
    const move_iterator<Iter2>& y) -> decltype(x.base() - y.base());
template <RandomAccessIterator Iterator>
  move_iterator<Iterator> operator+(
    Iter::difference_type n, const move_iterator<Iterator>& x);
template <InputIterator Iter>
  move_iterator<Iter> make_move_iterator(const Iterator& i);
template<InputIterator Iter>
  concept_map InputIterator<move_iterator<Iter> > { }
template<ForwardIterator Iter>
  concept_map ForwardIterator<move_iterator<Iter> > { }
template<BidirectionalIterator Iter>
  concept_map BidirectionalIterator<move_iterator<Iter> > { }
template<RandomAccessIterator Iter>
  concept_map RandomAccessIterator<move_iterator<Iter> > { }

// 24.5, stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);

template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator;

template<class charT, class traits = char_traits<charT> >
  class istreambuf_iterator;
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT,traits>& a,
          const istreambuf_iterator<charT,traits>& b);
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT,traits>& a,
```

```
            const istreambuf_iterator<charT,traits>& b);

  template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator;
}
```

24.3 Iterator operations [iterator.operations]

1 Since only random access iterators provide + and - operators, the library provides two function templates advance and distance. These function templates use + and - for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

```
template <InputIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
template <BidirectionalIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
template <RandomAccessIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
```

2 *Requires:* n shall be negative only for bidirectional and random access iterators.

3 *Effects:* Increments (or decrements for negative n) iterator reference i by n.

```
template <InputIterator Iter>
  Iter::difference_type
  distance(Iter first, Iter last);
template <RandomAccessIterator Iter>
  Iter::difference_type distance(Iter first, Iter last);
```

4 *Effects:* Returns the number of increments or decrements needed to get from first to last.

5 *Requires:* last shall be reachable from first.

```
template <InputIterator Iter>
  Iter next(Iter x,
    Iter::difference_type n = 1);
```

6 *Effects:* Equivalent to `advance(x, n); return x;`

```
template <BidirectionalIterator Iter>
  Iter prev(Iter x,
    Iter::difference_type n = 1);
```

7 *Effects:* Equivalent to `advance(x, -n); return x;`

24.4 Predefined iterators [predef.iterators]

24.4.1 Reverse iterators [reverse.iterators]

1 Bidirectional and random access iterators have corresponding reverse iterator adaptors that iterate through the data structure in the opposite direction. They have the same signatures as the corresponding iterators. The fundamental relation between a reverse iterator and its corresponding iterator i is established by the identity: `&*(reverse_iterator(i)) == &*(i - 1)`.

2 This mapping is dictated by the fact that while there is always a pointer past the end of an array, there might not be a valid pointer before the beginning of an array.

24.4.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  template <BidirectionalIterator Iter>
  class reverse_iterator {
  protected:
    Iter current;
  public:
    typedef Iter iterator_type;
    typedef Iter::value_type value_type;
    typedef Iter::difference_type difference_type;
    typedef Iter::reference reference;
    typedef Iter::pointer pointer;

    reverse_iterator();
    explicit reverse_iterator(Iter x);
    template <class U>
      requires HasConstructor<Iter, const U&>}
      reverse_iterator(const reverse_iterator<U>& u);
    template <class U>
      requires HasAssign<Iter, const U&>
      reverse_iterator operator=(const reverse_iterator<U>& u);

    Iter base() const;      // explicit
    reference operator*() const;
    pointer   operator->() const;

    reverse_iterator& operator++();
    reverse_iterator  operator++(int);
    reverse_iterator& operator--();
    reverse_iterator  operator--(int);

    requires RandomAccessIterator<Iter> reverse_iterator  operator+ (difference_type n) const;
    requires RandomAccessIterator<Iter> reverse_iterator& operator+=(difference_type n);
    requires RandomAccessIterator<Iter> reverse_iterator  operator- (difference_type n) const;
    requires RandomAccessIterator<Iter> reverse_iterator& operator-=(difference_type n);
    requires RandomAccessIterator<Iter> unspecified  operator[](difference_type n) const;
  };

  template <BidirectionalIterator Iter1, BidirectionalIterator Iter2>
    requires HasEqualTo<Iter1, Iter2>
    bool operator==(
      const reverse_iterator<Iter1>& x,
      const reverse_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasGreater<Iter1, Iter2>
    bool operator<(
      const reverse_iterator<Iter1>& x,
      const reverse_iterator<Iter2>& y);
  template <BidirectionalIterator Iter1, BidirectionalIterator Iter2>
    requires HasNotEqualTo<Iter1, Iter2>
    bool operator!=(
      const reverse_iterator<Iter1>& x,
```

```
      const reverse_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasLess<Iter1, Iter2>
    bool operator>(
      const reverse_iterator<Iter1>& x,
      const reverse_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasLessEqual<Iter1, Iter2>
    bool operator>=(
      const reverse_iterator<Iter1>& x,
      const reverse_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasGreaterEqual<Iter1, Iter2>
    bool operator<=(
      const reverse_iterator<Iter1>& x,
      const reverse_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasMinus<Iter2, Iter1>
    auto operator-(
      const reverse_iterator<Iter1>& x,
      const reverse_iterator<Iter2>& y) -> decltype(y.base() - x.base());
  template <RandomAccessIterator Iterator>
    reverse_iterator<Iter> operator+(
      Iter::difference_type n,
      const reverse_iterator<Iter>& x);

  template<BidirectionalIterator Iter>
  concept_map BidirectionalIterator<reverse_iterator<Iter> > { }

  template<RandomAccessIterator Iter>
  concept_map RandomAccessIterator<reverse_iterator<Iter> > { }
}
```

24.4.1.2 `reverse_iterator` operations [reverse.iter.ops]

24.4.1.2.1 `reverse_iterator` constructor [reverse.iter.cons]

```
reverse_iterator();
```

1 *Effects:* Default initializes `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a default constructed iterator of type `Iterator`.

```
explicit reverse_iterator(Iter x);
```

2 *Effects:* Initializes `current` with `x`.

```
template <class U>
  requires HasConstructor<Iter, const U&>
  reverse_iterator(const reverse_iterator<U> &u);
```

3 *Effects:* Initializes `current` with `u.current`.

24.4.1.2.2 `reverse_iterator::operator=` [reverse.iter.op=]

```
template <class U>
```

```
  requires HasAssign<Iter, const U&>
  reverse_iterator&
    operator=(const reverse_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to current.

2 *Returns:* `*this`.

24.4.1.2.3 Conversion [reverse.iter.conv]

```
Iter base() const;          // explicit
```

1 *Returns:* `current`.

24.4.1.2.4 operator* [reverse.iter.op.star]

```
reference operator*() const;
```

1 *Effects:*

```
  this->tmp = current;
  --this->tmp;
  return *this->tmp;
```

2 [*Note:* This operation must use an auxiliary member variable, rather than a temporary variable, to avoid returning a reference that persists beyond the lifetime of its associated iterator. (See 24.1.) The name of this member variable is shown for exposition only. — *end note*]

24.4.1.2.5 operator-> [reverse.iter.opref]

```
pointer operator->() const;
```

1 *Returns:*

```
  &(operator*());
```

24.4.1.2.6 operator++ [reverse.iter.op++]

```
reverse_iterator& operator++();
```

1 *Effects:* `--current;`

2 *Returns:* `*this`.

```
reverse_iterator operator++(int);
```

3 *Effects:*

```
  reverse_iterator tmp = *this;
  --current;
  return tmp;
```

24.4.1.2.7 operator-- [reverse.iter.op--]

```
reverse_iterator& operator--();
```

1 *Effects:* `++current`

2 *Returns:* `*this`.

```
reverse_iterator operator--(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
++current;
return tmp;
```

24.4.1.2.8 operator+ [reverse.iter.op+]

```
requires RandomAccessIterator<Iter>
reverse_iterator
operator+(difference_type n) const;
```

1 *Returns:* `reverse_iterator(current-n)`.

24.4.1.2.9 operator+= [reverse.iter.op+=]

```
requires RandomAccessIterator<Iter>
reverse_iterator&
operator+=(difference_type n);
```

1 *Effects:* `current -= n;`

2 *Returns:* `*this`.

24.4.1.2.10 operator- [reverse.iter.op-]

```
requires RandomAccessIterator<Iter>
reverse_iterator
operator-(difference_type n) const;
```

1 *Returns:* `reverse_iterator(current+n)`.

24.4.1.2.11 operator-= [reverse.iter.op-=]

```
requires RandomAccessIterator<Iter>
reverse_iterator&
operator-=(difference_type n);
```

1 *Effects:* `current += n;`

2 *Returns:* `*this`.

24.4.1.2.12 operator[] [reverse.iter.opindex]

```
requires RandomAccessIterator<Iter>
unspecified  operator[](
    difference_type n) const;
```

1 *Returns:* `current[-n-1]`.

24.4.1.2.13 operator== [reverse.iter.op==]

```
template <BidirectionalIterator Iter1, BidirectionalIterator Iter2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

1 *Returns:* `x.current == y.current`.

24.4.1.2.14 operator< [reverse.iter.op<]

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasGreater<Iter1, Iter2>
  bool operator<(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

1 *Returns:* `x.current > y.current`.

24.4.1.2.15 operator!= [reverse.iter.op!=]

```
template <BidirectionalIterator Iter1, BidirectionalIterator Iter2>
  requires HasNotEqualTo<Iter1, Iter2>
  bool operator!=(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

1 *Returns:* `x.current != y.current`.

24.4.1.2.16 operator> [reverse.iter.op>]

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter1, Iter2>
  bool operator>(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

1 *Returns:* `x.current < y.current`.

24.4.1.2.17 operator>= [reverse.iter.op>=]

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLessEqual<Iter1, Iter2>
  bool operator>=(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

1 *Returns:* `x.current <= y.current`.

24.4.1.2.18 operator<= [reverse.iter.op<=]

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasGreaterEqual<Iter1, Iter2>
  bool operator<=(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y);
```

1 *Returns:* `x.current >= y.current`.

24.4.1.2.19 operator- [reverse.iter.opdiff]

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasMinus<Iter2, Iter1>
  auto operator-(
    const reverse_iterator<Iter1>& x,
    const reverse_iterator<Iter2>& y) -> decltype(y.base() - x.base());
```

1 *Returns:* `y.current - x.current`.

24.4.1.2.20 operator+ [reverse.iter.opsum]

```
template <RandomAccessIterator Iterator>
  reverse_iterator<Iter> operator+(
    Iter::difference_type n,
    const reverse_iterator<Iter>& x);
```

1 *Returns:* `reverse_iterator<Iter> (x.current - n)`.

24.4.1.3 Concept maps [reverse.iter.maps]

```
template<BidirectionalIterator Iter>
concept_map BidirectionalIterator<reverse_iterator<Iter> > { }
```

1 *Note:* This concept map template states that reverse iterators are themselves bidirectional iterators.

```
template<RandomAccessIterator Iter>
concept_map RandomAccessIterator<reverse_iterator<Iter> > { }
```

2 *Note:* This concept map template states that reverse iterators are themselves random access iterators when the underlying iterator is a random access iterator.

24.4.2 Insert iterators [insert.iterators]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range `[first,last)` to be copied into a range starting with result. The same code with `result` being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite* mode.

2 An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. `operator*` returns the insert iterator itself. The assignment `operator=(const T& x)` is defined on insert iterators to allow writing into them, it inserts `x` right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. `back_insert_iterator` inserts elements at the end of a container, `front_insert_iterator` inserts elements at the beginning of a container, and `insert_iterator` inserts elements where the iterator points to in a container. `back_inserter`, `front_inserter`, and `inserter` are three functions making the insert iterators out of a container.

24.4.2.1 Class template `back_insert_iterator` [back.insert.iterator]

```
namespace std {
  template <BackInsertionContainer Cont>
  class back_insert_iterator {
  protected:
    Cont* container;

  public:
    typedef Cont container_type;
    typedef void                              value_type;
    typedef void                              difference_type;
    typedef back_insert_iterator<Cont>&       reference;
    typedef void                              pointer;

    explicit back_insert_iterator(Cont& x);
    requires CopyConstructible<Cont::value_type>
      back_insert_iterator<Cont>&
        operator=(const Cont::value_type& value);
    back_insert_iterator<Cont>&
      operator=(Cont::value_type&& value);

    back_insert_iterator<Cont>& operator*();
    back_insert_iterator<Cont>& operator++();
    back_insert_iterator<Cont>  operator++(int);
  };

  template <BackInsertionContainer Cont>
    back_insert_iterator<Cont> back_inserter(Cont& x);

  template<BackInsertionContainer Cont>
    concept_map Iterator<back_insert_iterator<Cont> > { }
}
```

24.4.2.2 `back_insert_iterator` operations [back.insert.iter.ops]

24.4.2.2.1 `back_insert_iterator` constructor [back.insert.iter.cons]

```
explicit back_insert_iterator(Cont& x);
```

1 *Effects:* Initializes `container` with `&x`.

24.4.2.2.2 `back_insert_iterator::operator=` [back.insert.iter.op=]

```
requires CopyConstructible<Cont::value_type>
```

```
back_insert_iterator<Cont>&
  operator=(const Cont::value_type& value);
```

1 *Effects:* `push_back(*container, Cont::value_type(value));`

2 *Returns:* `*this`.

```
back_insert_iterator<Cont>&
  operator=(Cont::value_type&& value);
```

3 *Effects:* `push_back(*container, std::move(value));`

4 *Returns:* `*this`.

24.4.2.2.3 back_insert_iterator::operator* [back.insert.iter.op*]

```
back_insert_iterator<Cont>& operator*();
```

1 *Returns:* `*this`.

24.4.2.2.4 back_insert_iterator::operator++ [back.insert.iter.op++]

```
back_insert_iterator<Cont>& operator++();
back_insert_iterator<Cont>  operator++(int);
```

1 *Returns:* `*this`.

24.4.2.2.5 back_inserter [back.inserter]

```
template <BackInsertionContainer Cont>
  back_insert_iterator<Cont> back_inserter(Cont& x);
```

1 *Returns:* `back_insert_iterator<Cont>(x)`.

24.4.2.2.6 Concept maps [back.insert.iter.maps]

```
template<BackInsertionContainer Cont>
  concept_map Iterator<back_insert_iterator<Cont> > { }
```

1 *Note:* Declares that `back_insert_iterator` is an iterator.

24.4.2.3 Class template front_insert_iterator [front.insert.iterator]

```
namespace std {
  template <FrontInsertionContainer Cont>
  class front_insert_iterator
  protected:
    Cont* container;

  public:
    typedef Cont container_type;
    typedef void                              value_type;
    typedef void                              difference_type;
    typedef front_insert_iterator<Cont>&      reference;
    typedef void                              pointer;
```

```
    explicit front_insert_iterator(Cont& x);
    requires CopyConstructible<Cont::value_type>
      front_insert_iterator<Cont>&
        operator=(const Cont::value_type& value);
    front_insert_iterator<Cont>&
      operator=(Cont::value_type&& value);

    front_insert_iterator<Cont>& operator*();
    front_insert_iterator<Cont>& operator++();
    front_insert_iterator<Cont>  operator++(int);
  };

  template <FrontInsertionContainer Cont>
    front_insert_iterator<Cont> front_inserter(Cont& x);

  template<FrontInsertionContainer Cont>
    concept_map Iterator<front_insert_iterator<Cont> > { }
}
```

24.4.2.4 `front_insert_iterator` operations [front.insert.iter.ops]

24.4.2.4.1 `front_insert_iterator` constructor [front.insert.iter.cons]

```
explicit front_insert_iterator(Cont& x);
```

1 *Effects:* Initializes `container` with `&x`.

24.4.2.4.2 `front_insert_iterator::operator=` [front.insert.iter.op=]

```
requires CopyConstructible<Cont::value_type>
  front_insert_iterator<Cont>&
    operator=(const Cont::value_type& value);
```

1 *Effects:* `push_front(*container, Cont::value_type(value));`

2 *Returns:* `*this`.

```
front_insert_iterator<Cont>&
  operator=(Cont::value_type&& value);
```

3 *Effects:* `push_front(*container, std::move(value));`

4 *Returns:* `*this`.

24.4.2.4.3 `front_insert_iterator::operator*` [front.insert.iter.op*]

```
front_insert_iterator<Cont>& operator*();
```

1 *Returns:* `*this`.

24.4.2.4.4 `front_insert_iterator::operator++` [front.insert.iter.op++]

```
front_insert_iterator<Cont>& operator++();
front_insert_iterator<Cont>  operator++(int);
```

1 *Returns:* `*this`.

24.4.2.4.5 `front_inserter` [front.inserter]

```
template <FrontInsertionContainer Cont>
  front_insert_iterator<Cont> front_inserter(Cont& x);
```

1 *Returns:* `front_insert_iterator<Cont>(x)`.

24.4.2.4.6 Concept maps [front.insert.iter.maps]

```
template<FrontInsertionContainer Cont>
  concept_map Iterator<front_insert_iterator<Cont> > { }
```

1 *Note:* Declares that `front_insert_iterator` is an iterator.

24.4.2.5 Class template `insert_iterator` [insert.iterator]

```
namespace std {
  template <InsertionContainer Cont>
  class insert_iterator {
  protected:
    Cont* container;
    Cont::iterator iter;

  public:
    typedef Cont container_type;
    typedef void                        value_type;
    typedef void                        difference_type;
    typedef insert_iterator<Cont>&      reference;
    typedef void                        pointer;

    insert_iterator(Cont& x, Cont::iterator i);
    requires CopyConstructible<Cont::value_type>
      insert_iterator<Cont>&
        operator=(const Cont::value_type& value);
    insert_iterator<Cont>&
      operator=(Cont::value_type&& value);

    insert_iterator<Cont>& operator*();
    insert_iterator<Cont>& operator++();
    insert_iterator<Cont>& operator++(int);
  };

  template <InsertionContainer Cont>
    insert_iterator<Cont> inserter(Cont& x, Cont::iterator i);

  template<InsertionContainer Cont>
    concept_map Iterator<insert_iterator<Cont> > { }
}
```

24.4.2.6 `insert_iterator` operations [insert.iter.ops]

24.4.2.6.1 `insert_iterator` constructor [insert.iter.cons]

```
insert_iterator(Cont& x, Cont::iterator i);
```

1 *Effects:* Initializes `container` with `&x` and `iter` with `i`.

24.4.2.6.2 insert_iterator::operator= [insert.iter.op=]

```
requires CopyConstructible<Cont::value_type>
  insert_iterator<Cont>&
    operator=(const Cont::value_type& value);
```

1 *Effects:*

```
iter = insert(*container, iter, Cont::value_type(value));
++iter;
```

2 *Returns:* `*this`.

```
insert_iterator<Cont>&
  operator=(Cont::value_type&& value);
```

3 *Effects:*

```
iter = insert(*container, iter, std::move(value));
++iter;
```

4 *Returns:* `*this`.

24.4.2.6.3 insert_iterator::operator* [insert.iter.op*]

```
insert_iterator<Cont>& operator*();
```

1 *Returns:* `*this`.

24.4.2.6.4 insert_iterator::operator++ [insert.iter.op++]

```
insert_iterator<Cont>& operator++();
insert_iterator<Cont>& operator++(int);
```

1 *Returns:* `*this`.

24.4.2.6.5 inserter [inserter]

```
template <InsertionContainer Cont>
  insert_iterator<Cont> inserter(Cont& x, Cont::iterator i);
```

1 *Returns:* `insert_iterator<Cont>(x, i)`.

24.4.2.6.6 Concept maps [insert.iter.maps]

```
template<InsertionContainer Cont>
  concept_map Iterator<insert_iterator<Cont> > { }
```

1 *Note:* Declares that `insert_iterator` is an iterator.

24.4.3 Move iterators [move.iterators]

1 Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its dereference operator implicitly converts the value returned by the underlying iterator's

dereference operator to an rvalue reference. Some generic algorithms can be called with move iterators to replace copying with moving.

2 [*Example:*

```
set<string> s;
// populate the set s
vector<string> v1(s.begin(), s.end());          // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // moves strings into v2
```

— *end example*]

24.4.3.1 Class template `move_iterator` [move.iterator]

```
namespace std {
  template <InputIterator Iter>
  class move_iterator {
  public:
    typedef Iter                                          iterator_type;
    typedef Iter::difference_type   difference_type;
    typedef Iterator                                          pointer;
    typedef Iter::value_type        value_type;
    typedef value_type&&                                      reference;

    move_iterator();
    explicit move_iterator(Iter i);
    template <class U>
      requires HasConstructor<Iter, const U&>
      move_iterator(const move_iterator<U>& u);
    template <class U>
      requires HasAssign<Iter, const U&>
      move_iterator& operator=(const move_iterator<U>& u);

    iterator_type base() const;
    reference operator*() const;
    pointer operator->() const;

    move_iterator& operator++();
    move_iterator operator++(int);
    requires BidirectionalIterator<Iter> move_iterator& operator--();
    requires BidirectionalIterator<Iter> move_iterator operator--(int);

    requires RandomAccessIterator<Iter> move_iterator operator+(difference_type n) const;
    requires RandomAccessIterator<Iter> move_iterator& operator+=(difference_type n);
    requires RandomAccessIterator<Iter> move_iterator operator-(difference_type n) const;
    requires RandomAccessIterator<Iter> move_iterator& operator-=(difference_type n);
    requires RandomAccessIterator<Iter>
      unspecified  operator[](difference_type n) const;

  private:
    Iter current;    // exposition only
  };

  template <InputIterator Iter1, InputIterator Iter2>
    requires HasEqualTo<Iter1, Iter2>
```

```
    bool operator==(
      const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
  template <InputIterator Iter1, InputIterator Iter2>
    requires HasEqualTo<Iter1, Iter2>
    bool operator!=(
      const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasLess<Iter1, Iter2>
    bool operator<(
      const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasLess<Iter2, Iter1>
    bool operator<=(
      const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasLess<Iter2, Iter1>
    bool operator>(
      const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasLess<Iter1, Iter2>
    bool operator>=(
      const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);

  template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
    requires HasMinus<Iter1, Iter2>
    auto operator-(
      const move_iterator<Iter1>& x,
      const move_iterator<Iter2>& y) -> decltype(x.base() - y.base());
  template <RandomAccessIterator Iter>
    move_iterator<Iter> operator+(
      Iter::difference_type n, const move_iterator<Iter>& x);
  template <InputIterator Iter>
    move_iterator<Iter> make_move_iterator(const Iter& i);

  template<InputIterator Iter>
    concept_map InputIterator<move_iterator<Iter> > { }
  template<ForwardIterator Iter>
    concept_map ForwardIterator<move_iterator<Iter> > { }
  template<BidirectionalIterator Iter>
    concept_map BidirectionalIterator<move_iterator<Iter> > { }
  template<RandomAccessIterator Iter>
    concept_map RandomAccessIterator<move_iterator<Iter> > { }
}
```

24.4.3.2 `move_iterator` operations [move.iter.ops]

24.4.3.2.1 `move_iterator` constructors [move.iter.op.const]

```
move_iterator();
```

1 *Effects:* Constructs a `move_iterator`, default initializing `current`.

```
explicit move_iterator(Iter i);
```

2 *Effects:* Constructs a `move_iterator`, intializing `current` with `i`.

```
template <class U>
  requires HasConstructor<Iter, const U&>
  move_iterator(const move_iterator<U>& u);
```

3 *Effects:* Constructs a `move_iterator`, initializing `current` with `u.base()`.

24.4.3.2.2 `move_iterator::operator=` [move.iter.op=]

```
template <class U>
  requires HasAssign<Iter, const U&>
  move_iterator& operator=(const move_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to `current`.

24.4.3.2.3 `move_iterator` conversion [move.iter.op.conv]

```
Iter base() const;
```

1 *Returns:* `current`.

24.4.3.2.4 `move_iterator::operator*` [move.iter.op.star]

```
reference operator*() const;
```

1 *Returns:* `*current`, implicitly converted to an rvalue reference.

24.4.3.2.5 `move_iterator::operator->` [move.iter.op.ref]

```
pointer operator->() const;
```

1 *Returns:* `current`.

24.4.3.2.6 `move_iterator::operator++` [move.iter.op.incr]

```
move_iterator& operator++();
```

1 *Effects:* `++current`.

2 *Returns:* `*this`.

```
move_iterator& operator++(int);
```

3 *Effects:*

```
move_iterator tmp = *this;
++current;
return tmp;
```

24.4.3.2.7 `move_iterator::operator--` [move.iter.op.decr]

```
requires BidirectionalIterator<Iter> move_iterator& operator--();
```

1 *Effects:* `--current`.

2 *Returns:* `*this`.

```
requires BidirectionalIterator<Iter> move_iterator& operator--(int);
```

3 *Effects:*

```
move_iterator tmp = *this;
--current;
return tmp;
```

24.4.3.2.8 `move_iterator::operator+` [move.iter.op.+]

```
requires RandomAccessIterator<Iter> move_iterator operator+(difference_type n) const;
```

1 *Returns:* `move_iterator(current + n)`.

24.4.3.2.9 `move_iterator::operator+=` [move.iter.op.+=]

```
requires RandomAccessIterator<Iter> move_iterator& operator+=(difference_type n);
```

1 *Effects:* `current += n`.

2 *Returns:* `*this`.

24.4.3.2.10 `move_iterator::operator-` [move.iter.op.-]

```
requires RandomAccessIterator<Iter> move_iterator operator-(difference_type n) const;
```

1 *Returns:* `move_iterator(current - n)`.

24.4.3.2.11 `move_iterator::operator-=` [move.iter.op.-=]

```
requires RandomAccessIterator<Iter> move_iterator& operator-=(difference_type n);
```

1 *Effects:* `current -= n`.

2 *Returns:* `*this`.

24.4.3.2.12 `move_iterator::operator[]` [move.iter.op.index]

```
requires RandomAccessIterator<Iter>
  unspecified  operator[](difference_type n) const;
```

1 *Returns:* `current[n]`, implicitly converted to an rvalue reference.

24.4.3.2.13 `move_iterator` comparisons [move.iter.op.comp]

```
template <InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
```

1 *Returns:* `x.base() == y.base()`.

```
template <InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator!=(const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
```

2 *Returns:* `!(x == y)`.

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter1, Iter2>
  bool operator<(const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
```

3 *Returns:* `x.base() < y.base()`.

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter2, Iter1>
  bool operator<=(const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
```

4 *Returns:* `!(y < x)`.

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter2, Iter1>
  bool operator>(const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
```

5 *Returns:* `y < x`.

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasLess<Iter1, Iter2>
  bool operator>=(const move_iterator<Iter1>& x, const move_iterator<Iter2>& y);
```

6 *Returns:* `!(x < y)`.

24.4.3.2.14 `move_iterator` non-member functions [move.iter.nonmember]

```
template <RandomAccessIterator Iter1, RandomAccessIterator Iter2>
  requires HasMinus<Iter1, Iter2>
  auto operator-(
    const move_iterator<Iter1>& x,
    const move_iterator<Iter2>& y) -> decltype(x.base() - y.base());
```

1 *Returns:* `x.base() - y.base()`.

```
template <RandomAccessIterator Iter>
  move_iterator<Iter> operator+(
    Iter::difference_type n, const move_iterator<Iter>& x);
```

2 *Returns:* `x + n`.

```
template <InputIterator Iter>
move_iterator<Iter> make_move_iterator(const Iter& i);
```

3 *Returns:* `move_iterator<Iter>(i)`.

24.4.3.2.15 Concept maps [move.iter.maps]

```
template<InputIterator Iter>
  concept_map InputIterator<move_iterator<Iter> > { }
```

1 *Note:* Declares that a `move_iterator` is an input iterator.

```
template<ForwardIterator Iter>
  concept_map ForwardIterator<move_iterator<Iter> > { }
```

2 *Note:* Declares that a `move_iterator` is a forward iterator if its underlying iterator is a forward iterator.

```
template<BidirectionalIterator Iter>
  concept_map BidirectionalIterator<move_iterator<Iter> > { }
```

3 *Note:* Declares that a `move_iterator` is a bidirectional iterator if its underlying iterator is a bidirectional iterator.

```
template<RandomAccessIterator Iter>
  concept_map RandomAccessIterator<move_iterator<Iter> > { }
```

4 *Note:* Declares that a `move_iterator` is a random access iterator if its underlying iterator is a random access iterator.

24.5 Stream iterators [stream.iterators]

1 To make it possible for algorithmic templates to work directly with input/output streams, appropriate iterator-like class templates are provided.

[*Example:*

```
partial_sum_copy(istream_iterator<double, char>(cin),
  istream_iterator<double, char>(),
  ostream_iterator<double, char>(cout, "\n"));
```

reads a file containing floating point numbers from `cin`, and prints the partial sums onto `cout`. — *end example*]

24.5.1 Class template `istream_iterator` [istream.iterator]

1 `istream_iterator` reads (using `operator>>`) successive elements from the input stream for which it was constructed. After it is constructed, and every time `++` is used, the iterator reads and stores a value of `T`. If the end of stream is reached (`operator void*()` on the stream returns `false`), the iterator becomes equal to the *end-of-stream* iterator value. The constructor with no arguments `istream_iterator()` always constructs an end of stream input iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end of stream is not defined. For any other iterator value a `const T&` is returned. The result of `operator->` on an end of stream is not defined. For any other iterator value a `const T*` is returned. It is impossible to store things into istream iterators. The main peculiarity of the istream iterators is the fact that `++` operators are not equality preserving, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time `++` is used a new value is read.

2 The practical consequence of this fact is that istream iterators can be used only for one-pass algorithms, which actually makes perfect sense, since for multi-pass algorithms it is always more appropriate to use in-memory data structures.

3 Two end-of-stream iterators are always equal. An end-of-stream iterator is not equal to a non-end-of-stream iterator. Two non-end-of-stream iterators are equal when they are constructed from the same stream.

```
namespace std {
  template <class T, class charT = char, class traits = char_traits<charT>,
      class Distance = ptrdiff_t>
  class istream_iterator:
    public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
  public:
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_istream<charT,traits> istream_type;
    istream_iterator();
```

```
    istream_iterator(istream_type& s);
    istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);
   ~istream_iterator();

    const T& operator*() const;
    const T* operator->() const;
    istream_iterator<T,charT,traits,Distance>& operator++();
    istream_iterator<T,charT,traits,Distance>  operator++(int);
  private:
    // basic_istream<charT,traits>* in_stream;                    exposition only
    // T value;                                                   exposition only
  };

  template <class T, class charT, class traits, class Distance>
    bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
             const istream_iterator<T,charT,traits,Distance>& y);
  template <class T, class charT, class traits, class Distance>
    bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
             const istream_iterator<T,charT,traits,Distance>& y);
}
```

24.5.1.1 `istream_iterator` constructors and destructor [istream.iterator.cons]

`istream_iterator();`

1 *Effects:* Constructs the end-of-stream iterator.

`istream_iterator(istream_type& s);`

2 *Effects:* Initializes *in_stream* with `s`. *value* may be initialized during construction or the first time it is referenced.

`istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);`

3 *Effects:* Constructs a copy of `x`.

`~istream_iterator();`

4 *Effects:* The iterator is destroyed.

24.5.1.2 `istream_iterator` operations [istream.iterator.ops]

`const T& operator*() const;`

1 *Returns: value.*

`const T* operator->() const;`

2 *Returns:* `&(operator*())`.

`istream_iterator<T,charT,traits,Distance>& operator++();`

3 *Effects:* `*in_stream >>value` .

4 *Returns:* `*this`.

`istream_iterator<T,charT,traits,Distance> operator++(int);`

5 *Effects:*

```
    istream_iterator<T,charT,traits,Distance> tmp = *this;
    *in_stream >> value;
    return (tmp);

template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance> &x,
                  const istream_iterator<T,charT,traits,Distance> &y);
```

6 *Returns:* (x.*in_stream* == y.*in_stream*).

```
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance> &x,
                  const istream_iterator<T,charT,traits,Distance> &y);
```

7 *Returns:* !(x == y)

24.5.2 Class template ostream_iterator [ostream.iterator]

1 ostream_iterator writes (using operator<<) successive elements onto the output stream from which it was constructed. If it was constructed with char* as a constructor argument, this string, called a *delimiter string*, is written to the stream after every T is written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

2 ostream_iterator is defined as:

```
namespace std {
  template <class T, class charT = char, class traits = char_traits<charT> >
  class ostream_iterator:
    public iterator<output_iterator_tag, void, void, void, void> {
  public:
    typedef charT char_type;
    typedef traits traits_type;
    typedef basic_ostream<charT,traits> ostream_type;
    ostream_iterator(ostream_type& s);
    ostream_iterator(ostream_type& s, const charT* delimiter);
    ostream_iterator(const ostream_iterator<T,charT,traits>& x);
   ~ostream_iterator();
    ostream_iterator<T,charT,traits>& operator=(const T& value);

    ostream_iterator<T,charT,traits>& operator*();
    ostream_iterator<T,charT,traits>& operator++();
    ostream_iterator<T,charT,traits>& operator++(int);
  private:
    // basic_ostream<charT,traits>* out_stream;          exposition only
    // const charT* delim;                               exposition only
  };
}
```

24.5.2.1 ostream_iterator constructors and destructor [ostream.iterator.cons.des]

```
ostream_iterator(ostream_type& s);
```

1 *Effects:* Initializes *out_stream* with s and *delim* with null.

```
ostream_iterator(ostream_type& s, const charT* delimiter);
```

2 *Effects:* Initializes *out_stream* with `s` and *delim* with `delimiter`.

```
ostream_iterator(const ostream_iterator& x);
```

3 *Effects:* Constructs a copy of `x`.

```
~ostream_iterator();
```

4 *Effects:* The iterator is destroyed.

24.5.2.2 `ostream_iterator` operations [ostream.iterator.ops]

```
ostream_iterator& operator=(const T& value);
```

1 *Effects:*

```
*out_stream << value;
if(delim != 0)
  *out_stream << delim;
return (*this);
```

```
ostream_iterator& operator*();
```

2 *Returns:* `*this`.

```
ostream_iterator& operator++();
ostream_iterator& operator++(int);
```

3 *Returns:* `*this`.

24.5.3 Class template `istreambuf_iterator` [istreambuf.iterator]

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class istreambuf_iterator
     : public iterator<input_iterator_tag, charT,
                       typename traits::off_type, charT*, charT> {
  public:
    typedef charT                          char_type;
    typedef traits                         traits_type;
    typedef typename traits::int_type      int_type;
    typedef basic_streambuf<charT,traits> streambuf_type;
    typedef basic_istream<charT,traits>   istream_type;

    class proxy;                                          // exposition only

    public:
      istreambuf_iterator() throw();
      istreambuf_iterator(istream_type& s) throw();
      istreambuf_iterator(streambuf_type* s) throw();
      istreambuf_iterator(const proxy& p) throw();
      charT operator*() const;
      istreambuf_iterator<charT,traits>& operator++();
      proxy operator++(int);
      bool equal(istreambuf_iterator& b) const;
```

```
    private:
      streambuf_type* sbuf_;                                    // exposition only
  };

  template <class charT, class traits>
    bool operator==(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
  template <class charT, class traits>
    bool operator!=(const istreambuf_iterator<charT,traits>& a,
            const istreambuf_iterator<charT,traits>& b);
}
```

1 The class template `istreambuf_iterator` reads successive *characters* from the streambuf for which it was constructed. `operator*` provides access to the current input character, if any. Each time `operator++` is evaluated, the iterator advances to the next input character. If the end of stream is reached (streambuf_type::sgetc() returns `traits::eof()`), the iterator becomes equal to the *end of stream* iterator value. The default constructor `istreambuf_iterator()` and the constructor `istreambuf_iterator(0)` both construct an end of stream iterator object suitable for use as an end-of-range.

2 The result of `operator*()` on an end of stream is undefined. For any other iterator value a `char_type` value is returned. It is impossible to assign a character via an input iterator.

3 Note that in the input iterators, `++` operators are not *equality preserving*, that is, `i == j` does not guarantee at all that `++i == ++j`. Every time `++` is evaluated a new value is used.

4 The practical consequence of this fact is that an `istreambuf_iterator` object can be used only for *one-pass algorithms*. Two end of stream iterators are always equal. An end of stream iterator is not equal to a non-end of stream iterator.

24.5.3.1 Class template `istreambuf_iterator::proxy` [istreambuf.iterator::proxy]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class istreambuf_iterator<charT, traits>::proxy {
    charT keep_;
    basic_streambuf<charT,traits>* sbuf_;
    proxy(charT c,
      basic_streambuf<charT,traits>* sbuf);
      : keep_(c), sbuf_(sbuf) { }
  public:
    charT operator*() { return keep_; }
  };
}
```

1 Class `istreambuf_iterator<charT,traits>::proxy` is for exposition only. An implementation is permitted to provide equivalent functionality without providing a class with this name. Class `istreambuf_iterator<charT, traits>::proxy` provides a temporary placeholder as the return value of the post-increment operator (`operator++`). It keeps the character pointed to by the previous value of the iterator for some possible future access to get the character.

24.5.3.2 `istreambuf_iterator` constructors [istreambuf.iterator.cons]

```
istreambuf_iterator() throw();
```

1 *Effects:* Constructs the end-of-stream iterator.

```
istreambuf_iterator(basic_istream<charT,traits>& s) throw();
istreambuf_iterator(basic_streambuf<charT,traits>* s) throw();
```

2 *Effects:* Constructs an `istreambuf_iterator<>` that uses the `basic_streambuf<>` object `*(s.rdbuf())`, or `*s`, respectively. Constructs an end-of-stream iterator if `s.rdbuf()` is null.

```
istreambuf_iterator(const proxy& p) throw();
```

3 *Effects:* Constructs a `istreambuf_iterator<>` that uses the `basic_streambuf<>` object pointed to by the `proxy` object's constructor argument `p`.

24.5.3.3 `istreambuf_iterator::operator*` [istreambuf.iterator::op*]

```
charT operator*() const
```

1 *Returns:* The character obtained via the `streambuf` member *`sbuf_`*`->sgetc()`.

24.5.3.4 `istreambuf_iterator::operator++` [istreambuf.iterator::op++]

```
istreambuf_iterator<charT,traits>&
    istreambuf_iterator<charT,traits>::operator++();
```

1 *Effects:* *`sbuf_`*`->sbumpc()`.

2 *Returns:* `*this`.

```
proxy istreambuf_iterator<charT,traits>::operator++(int);
```

3 *Returns:* `proxy(`*`sbuf_`*`->sbumpc(), `*`sbuf_`*`)`.

24.5.3.5 `istreambuf_iterator::equal` [istreambuf.iterator::equal]

```
bool equal(istreambuf_iterator<charT,traits>& b) const;
```

1 *Returns:* `true` if and only if both iterators are at end-of-stream, or neither is at end-of-stream, regardless of what `streambuf` object they use.

24.5.3.6 `operator==` [istreambuf.iterator::op==]

```
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT,traits>& a,
                  const istreambuf_iterator<charT,traits>& b);
```

1 *Returns:* `a.equal(b)`.

24.5.3.7 `operator!=` [istreambuf.iterator::op!=]

```
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT,traits>& a,
                  const istreambuf_iterator<charT,traits>& b);
```

1 *Returns:* `!a.equal(b)`.

24.5.4 Class template `ostreambuf_iterator` [ostreambuf.iterator]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class ostreambuf_iterator :
    public iterator<output_iterator_tag, void, void, void, void> {
  public:
    typedef charT                          char_type;
    typedef traits                         traits_type;
    typedef basic_streambuf<charT,traits> streambuf_type;
    typedef basic_ostream<charT,traits>   ostream_type;

  public:
    ostreambuf_iterator(ostream_type& s) throw();
    ostreambuf_iterator(streambuf_type* s) throw();
    ostreambuf_iterator& operator=(charT c);

    ostreambuf_iterator& operator*();
    ostreambuf_iterator& operator++();
    ostreambuf_iterator& operator++(int);
    bool failed() const throw();

  private:
    // streambuf_type* sbuf_;                                    exposition only
  };
}
```

1 The class template `ostreambuf_iterator` writes successive *characters* onto the output stream from which it was constructed. It is not possible to get a character value out of the output iterator.

24.5.4.1 `ostreambuf_iterator` constructors [ostreambuf.iter.cons]

```
ostreambuf_iterator(ostream_type& s) throw();
```

1 *Requires:* `s.rdbuf()` shall not null pointer.

2 *Effects:* `:sbuf_(s.rdbuf()) {}`.

```
ostreambuf_iterator(streambuf_type* s) throw();
```

3 *Requires:* `s` shall not be a null pointer.

4 *Effects:* `:` *`sbuf_`* `(s) {}`.

24.5.4.2 `ostreambuf_iterator` operations [ostreambuf.iter.ops]

```
ostreambuf_iterator<charT,traits>&
  operator=(charT c);
```

1 *Effects:* If `failed()` yields `false`, calls *`sbuf_`*`->sputc(c)`; otherwise has no effect.

2 *Returns:* `*this`.

```
ostreambuf_iterator<charT,traits>& operator*();
```

3 *Returns:* `*this`.

```
ostreambuf_iterator<charT,traits>& operator++();
```

```
ostreambuf_iterator<charT,traits>& operator++(int);
```

4 *Returns:* `*this`.

```
bool failed() const throw();
```

5 *Returns:* `true` if in any prior use of member `operator=`, the call to `sbuf_->sputc()` returned `traits::eof()`; or `false` otherwise.

25 Algorithms library [algorithms]

1 This clause describes components that C++ programs may use to perform algorithmic operations on containers (clause 23) and other sequences.

2 The following subclauses describe components for non-modifying sequence operation, modifying sequence operations, sorting and related operations, and algorithms from the ISO C library, as summarized in Table 90.

Table 90 — Algorithms library summary

Subclause		Header(s)
25.1	Non-modifying sequence operations	
25.2	Mutating sequence operations	<algorithm>
25.3	Sorting and related operations	
25.4	C library algorithms	<cstdlib>

Header <algorithm> synopsis

```
namespace std {
  // 25.1, non-modifying sequence operations:
  template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    bool all_of(Iter first, Iter last, Pred pred);
  template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    bool any_of(Iter first, Iter last, Pred pred);
  template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    bool none_of(Iter first, Iter last, Pred pred);

  template<InputIterator Iter, Callable<auto, Iter::reference> Function>
    requires CopyConstructible<Function>
    Function for_each(Iter first, Iter last, Function f);
  template<InputIterator Iter, class T>
    requires HasEqualTo<Iter::value_type, T>
    Iter find(Iter first, Iter last, const T& value);
  template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    Iter find_if(Iter first, Iter last, Pred pred);
  template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
    requires CopyConstructible<Pred>
    Iter find_if_not(Iter first, Iter last, Pred pred);
  template<ForwardIterator Iter1, ForwardIterator Iter2>
    requires HasEqualTo<Iter1::value_type, Iter2::value_type>
    Iter1 find_end(Iter1 first1, Iter1 last1,
                   Iter2 first2, Iter2 last2);
  template<ForwardIterator Iter1, ForwardIterator Iter2,
           Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
    requires CopyConstructible<Pred>
    Iter1 find_end(Iter1 first1, Iter1 last1,
```

```
                Iter2 first2, Iter2 last2,
                Pred pred);

template<InputIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);
template<InputIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2,
                      Pred pred);

template<ForwardIterator Iter>
  requires EqualityComparable<Iter::value_type>
  Iter adjacent_find(Iter first, Iter last);
template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter adjacent_find(Iter first, Iter last, Pred pred);

template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::value_type, T>
  Iter::difference_type count(Iter first, Iter last, const T& value);
template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter::difference_type count_if(Iter first, Iter last, Pred pred);

template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2);
template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2, Pred pred);

template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2);
template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2, Pred pred);

template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2);
template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
```

```
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2,
               Pred pred);
template<ForwardIterator Iter, class T>
  requires HasEqualTo<Iter::value_type, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);
template<ForwardIterator Iter, class T,
         Predicate<auto, Iter::value_type, T> Pred>
  requires CopyConstructible<Pred>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value, Pred pred);

// 25.2, modifying sequence operations:
// 25.2.1, copy:
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy(InIter first, InIter last,
               OutIter result);
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy_n(InIter first, InIter::difference_type n,
                 OutIter result);
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter copy_if(InIter first, InIter last,
                  OutIter result, Pred pred);
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires OutputIterator<OutIter, InIter::reference>
  OutIter copy_backward(InIter first, InIter last,
                        OutIter result);

// 25.2.2, move:
template<InputIterator InIter, typename OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move(InIter first, InIter last,
               OutIter result);
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move_backward(InIter first, InIter last,
                        OutIter result);

// 25.2.3, swap:
template<class T>
  requires MoveAssignable<T> && MoveConstructible<T>
  void swap(T& a, T& b);
template<ValueType T, size_t N>
  requires Swappable<T>
  void swap(T (&a)[N], T (&b)[N]);
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  Iter2 swap_ranges(Iter1 first1, Iter1 last1,
                    Iter2 first2);
template<Iterator Iter1, Iterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  void iter_swap(Iter1 a, Iter2 b);
```

```
template<InputIterator InIter, class OutIter,
         Callable<auto, const InIter::value_type&> Op>
  requires OutputIterator<OutIter, Op::result_type>
        && CopyConstructible<Op>
  OutIter transform(InIter first, InIter last,
                    OutIter result, Op op);
template<InputIterator InIter1, InputIterator InIter2,
         class OutIter,
         Callable<auto, const InIter1::value_type&,
                        const InIter2::value_type&> BinaryOp>
  requires OutputIterator<OutIter, BinaryOp::result_type>
        && CopyConstructible<BinaryOp>
  OutIter transform(InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result,
                    BinaryOp binary_op);

template<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter, const T&>
        && HasEqualTo<Iter::value_type, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T& new_value);
template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter::reference, const T&>
        && CopyConstructible<Pred>
  void replace_if(Iter first, Iter last,
                  Pred pred, const T& new_value);
template<InputIterator InIter, typename OutIter, class T>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const T&>
        && HasEqualTo<InIter::value_type, T>
  OutIter replace_copy(InIter first, InIter last,
                       OutIter result,
                       const T& old_value, const T& new_value);
template<InputIterator InIter, typename OutIter,
         Predicate<auto, InIter::value_type> Pred, class T>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const T&>
        && CopyConstructible<Pred>
  OutIter replace_copy_if(InIter first, InIter last,
                          OutIter result,
                          Pred pred, const T& new_value);

template<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, const T&>
  void fill(Iter first, Iter last, const T& value);
template<class Iter, IntegralLike Size, class T>
  requires OutputIterator<Iter, const T&>
  void fill_n(Iter first, Size n, const T& value);

template<ForwardIterator Iter, Callable Generator>
  requires OutputIterator<Iter, Generator::result_type>
        && CopyConstructible<Generator>
```

```
  void generate(Iter first, Iter last,
                Generator gen);
template<class Iter, IntegralLike Size, Callable Generator>
  requires OutputIterator<Iter, Generator::result_type>
        && CopyConstructible<Generator>
  void generate_n(Iter first, Size n, Generator gen);

template<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && HasEqualTo<Iter::value_type, T>
  Iter remove(Iter first, Iter last,
              const T& value);
template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && CopyConstructible<Pred>
  Iter remove_if(Iter first, Iter last,
                 Pred pred);
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter, class T>
  requires HasEqualTo<InIter::value_type, T>
  OutIter remove_copy(InIter first, InIter last,
                      OutIter result, const T& value);
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter remove_copy_if(InIter first, InIter last,
                         OutIter result, Pred pred);

template<ForwardIterator Iter>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && EqualityComparable<Iter::value_type>
  Iter unique(Iter first, Iter last);
template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && CopyConstructible<Pred>
  Iter unique(Iter first, Iter last,
              Pred pred);
  template<InputIterator InIter, class OutIter>
    requires OutputIterator<OutIter, RvalueOf<InIter::value_type>::type>
        && EqualityComparable<InIter::value_type>
        && HasAssign<InIter::value_type, InIter::reference>
        && Constructible<InIter::value_type, InIter::reference>
    OutIter unique_copy(InIter first, InIter last, OutIter result);
  template<InputIterator InIter, class OutIter,
    EquivalenceRelation<auto, InIter::value_type> Pred>
    requires OutputIterator<OutIter, RvalueOf<InIter::value_type>::type>
        && HasAssign<InIter::value_type, InIter::reference>
        && Constructible<InIter::value_type, InIter::reference>
        && CopyConstructible<Pred>
    OutIter unique_copy(InIter first, InIter last, OutIter result, Pred pred);

template<BidirectionalIterator Iter>
  requires HasSwap<Iter::reference, Iter::reference>
  void reverse(Iter first, Iter last);
template<BidirectionalIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter reverse_copy(InIter first, InIter last, OutIter result);
```

```
template<ShuffleIterator Iter>
  Iter rotate(Iter first, Iter middle,
              Iter last);
template<ForwardIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter rotate_copy(InIter first, InIter middle,
                      InIter last, OutIter result);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
  void random_shuffle(Iter first,
                      Iter last);
template<RandomAccessIterator Iter, Callable<auto, Iter::difference_type> Rand>
  requires ShuffleIterator<Iter>
        && Convertible<Rand::result_type, Iter::difference_type>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand&& rand);

concept UniformRandomNumberGenerator<typename Rand> { }
template<RandomAccessIterator Iter, UniformRandomNumberGenerator Rand>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand&& g);

// 25.2.13, partitions:
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool is_partitioned(Iter first, Iter last, Pred pred);

template<BidirectionalIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter partition(Iter first, Iter last, Pred pred);
template<BidirectionalIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter stable_partition(Iter first, Iter last, Pred pred);
template <InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter1,
          OutputIterator<auto, InIter::reference> OutIter2, Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  pair<OutIter1, OutIter2>
  partition_copy(InIter first, InIter last,
                 OutIter1 out_true, OutIter2 out_false,
                 Pred pred);
template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter partition_point(Iter first, Iter last, Pred pred);

// 25.3, sorting and related operations:
// 25.3.1, sorting:
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort(Iter first, Iter last);
```

```
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void stable_sort(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void stable_sort(Iter first, Iter last,
                   Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last,
                    Compare comp);
template<InputIterator InIter, RandomAccessIterator RAIter>
  requires ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && HasLess<InIter::value_type, RAIter::value_type>
        && LessThanComparable<RAIter::value_type>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last);
template<InputIterator InIter, RandomAccessIterator RAIter, class Compare>
  requires ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && Predicate<Compare, InIter::value_type, RAIter::value_type>
        && StrictWeakOrder<Compare, RAIter::value_type>}
        && CopyConstructible<Compare>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last,
                           Compare comp);

template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_sorted(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
```

```
  bool is_sorted(Iter first, Iter last,
                 Compare comp);
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter is_sorted_until(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter is_sorted_until(Iter first, Iter last,
                       Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void nth_element(Iter first, Iter nth,
                   Iter last,  Compare comp);

// 25.3.3, binary search:
template<ForwardIterator Iter, class T>
  requires HasLess<Iter::value_type, T>
  Iter lower_bound(Iter first, Iter last,
                   const T& value);
template<ForwardIterator Iter, class T, Predicate<auto, Iter::value_type, T> Compare>
  requires CopyConstructible<Compare>
  Iter lower_bound(Iter first, Iter last,
                   const T& value, Compare comp);

template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::value_type>
  Iter upper_bound(Iter first, Iter last,
                   const T& value);
template<ForwardIterator Iter, class T, Predicate<auto, T, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter upper_bound(Iter first, Iter last,
                   const T& value, Compare comp);

template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::value_type>
        && HasLess<Iter::value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value);
template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value,
                Compare comp);
```

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::value_type>
        && HasLess<Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value);
template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value, Compare comp);

// 25.3.4, merge:
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         Predicate<auto, InIter2::value_type, InIter1::value_type> Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && CopyConstructible<Compare>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result, Compare comp);

template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);
template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last, Compare comp);

// 25.3.5, set operations:
template<InputIterator Iter1, InputIterator Iter2>
  requires HasLess<Iter1::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2);
template<InputIterator Iter1, InputIterator Iter2,
         typename Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::value_type>
```

```
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2,
                Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_union(InIter1 first1, InIter1 last1,
                    InIter2 first2, InIter2 last2,
                    OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter, CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_union(InIter1 first1, InIter1 last1,
                    InIter2 first2, InIter2 last2,
                    OutIter result, Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter, CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result, Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
```

```
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result, Compare comp);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                   InIter2 first2, InIter2 last2,
                                   OutIter result);
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter, CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                   InIter2 first2, InIter2 last2,
                                   OutIter result, Compare comp);

// 25.3.6, heap operations:
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void push_heap(Iter first, Iter last,
                 Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void pop_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void pop_heap(Iter first, Iter last,
                Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void make_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
```

```
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void make_heap(Iter first, Iter last,
                 Compare comp);

template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort_heap(Iter first, Iter last,
                 Compare comp);

template<RandomAccessIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_heap(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  bool is_heap(Iter first, Iter last, Compare comp);
template<RandomAccessIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter is_heap_until(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter is_heap_until(Iter first, Iter last,
                     Compare comp);

// 25.3.7, minimum and maximum:
template<LessThanComparable T> const T& min(const T& a, const T& b);
template<class T, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& min(const T& a, const T& b, Compare comp);
template<class T>
  T min(initializer_list<T> t);
template<class T, class Compare>
  T min(initializer_list<T> t, Compare comp);

template<LessThanComparable T> const T& max(const T& a, const T& b);
template<class T, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& max(const T& a, const T& b, Compare comp);
template<class T>
  T max(initializer_list<T> t);
template<class T, class Compare>
  T max(initializer_list<T> t, Compare comp);

template<LessThanComparable T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
template<class T>
```

```
  pair<const T&, const T&> minmax(initializer_list<T> t);
template<class T, class Compare>
  pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);

template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter min_element(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter min_element(Iter first, Iter last,
                   Compare comp);

template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter max_element(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter max_element(Iter first, Iter last,
                   Compare comp);

template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last, Compare comp);

template<InputIterator Iter1, InputIterator Iter2>
  requires HasLess<Iter1::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2, CopyConstructible Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::value_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2,
                               Compare comp);

// 25.3.9, permutations:
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  bool next_permutation(Iter first, Iter last);
template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  bool next_permutation(Iter first, Iter last, Compare comp);
```

```
  template<BidirectionalIterator Iter>
    requires ShuffleIterator<Iter>
          && LessThanComparable<Iter::value_type>
    bool prev_permutation(Iter first, Iter last);
  template<BidirectionalIterator Iter,
           StrictWeakOrder<auto, Iter::value_type> Compare>
    requires ShuffleIterator<Iter>
          && CopyConstructible<Compare>
    bool prev_permutation(Iter first, Iter last, Compare comp);
}
```

3 All of the algorithms are separated from the particular implementations of data structures and are parameterized by iterator types. Because of this, they can work with program-defined data structures, as long as these data structures have iterator types satisfying the assumptions on the algorithms.

4 Both in-place and copying versions are provided for certain algorithms.[263] When such a version is provided for *algorithm* it is called *algorithm*`_copy`. Algorithms that take predicates end with the suffix `_if` (which follows the suffix `_copy`).

5 [*Note:* Unless otherwise specified, algorithms that take function objects as arguments are permitted to copy those function objects freely. Programmers for whom object identity is important should consider using a wrapper class that points to a noncopied implementation object, or some equivalent solution. — *end note*]

6 In the description of the algorithms operators `+` and `-` are used with iterators that do not necessarily define these operators. In these cases the semantics of `a+n` is the same as that of

```
{ X tmp = a;
  advance(tmp, n);
  return tmp;
}
```

and that of `b-a` is the same as of

```
return distance(a, b);
```

25.1 Non-modifying sequence operations [alg.nonmodifying]

25.1.1 All of [alg.all_of]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool all_of(Iter first, Iter last, Pred pred);
```

1 *Returns:* `true` if `pred(*i)` is `true` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate.

25.1.2 Any of [alg.any_of]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool any_of(Iter first, Iter last, Pred pred);
```

263) The decision whether to include a copying version was usually based on complexity considerations. When the cost of doing the operation dominates the cost of copy, the copying version is not included. For example, `sort_copy` is not included because the cost of sorting is much more significant, and users might as well do `copy` followed by `sort`.

1 *Returns:* `true` if there exists an iterator `i` in the range `[first,last)` such that `pred(*i)` is `true`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate.

25.1.3 None of [alg.none_of]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool none_of(Iter first, Iter last, Pred pred);
```

1 *Returns:* `true` if `pred(*i)` is `false` for every iterator `i` in the range `[first,last)`, and `false` otherwise.

2 *Complexity:* At most `last - first` applications of the predicate.

25.1.4 For each [alg.foreach]

```
template<InputIterator Iter, Callable<auto, Iter::reference> Function>
  requires CopyConstructible<Function>
  Function for_each(Iter first, Iter last, Function f);
```

1 *Effects:* Applies `f` to the result of dereferencing every iterator in the range `[first,last)`, starting from `first` and proceeding to `last - 1`.

2 *Returns:* `f`.

3 *Complexity:* Applies `f` exactly `last - first` times.

25.1.5 Find [alg.find]

```
template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::value_type, T>
  Iter find(Iter first, Iter last, const T& value);

template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter find_if(Iter first, Iter last, Pred pred);

template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter find_if_not(Iter first, Iter last, Pred pred);
```

1 *Returns:* The first iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value`, `pred(*i) != false`, `pred(*i) == false`. Returns `last` if no such iterator is found.

2 *Complexity:* At most `last - first` applications of the corresponding predicate.

25.1.6 Find End [alg.find.end]

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2);
```

```
template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_end(Iter1 first1, Iter1 last1,
                 Iter2 first2, Iter2 last2,
                 Pred pred);
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The last iterator `i` in the range `[first1,last1 - (last2 - first2))` such that for any non-negative integer `n < (last2 - first2)`, the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last2 - first2) * (last1 - first1 - (last2 - first2) + 1)` applications of the corresponding predicate.

25.1.7 Find First [alg.find.first.of]

```
template<InputIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2);

template<InputIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 find_first_of(Iter1 first1, Iter1 last1,
                      Iter2 first2, Iter2 last2,
                      Pred pred);
```

1 *Effects:* Finds an element that matches one of a set of values.

2 *Returns:* The first iterator `i` in the range `[first1,last1)` such that for some iterator `j` in the range `[first2,last2)` the following conditions hold: `*i == *j, pred(*i,*j) != false`. Returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last1-first1) * (last2-first2)` applications of the corresponding predicate.

25.1.8 Adjacent find [alg.adjacent.find]

```
template<ForwardIterator Iter>
  requires EqualityComparable<Iter::value_type>
  Iter adjacent_find(Iter first, Iter last);

template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter adjacent_find(Iter first, Iter last, Pred pred);
```

1 *Returns:* The first iterator `i` such that both `i` and `i + 1` are in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i + 1), pred(*i, *(i + 1)) != false`. Returns `last` if no such iterator is found.

2 *Complexity:* For a nonempty range, exactly `min((i - first) + 1, (last - first) - 1)` applications of the corresponding predicate, where `i` is `adjacent_find`'s return value.

25.1.9 Count [alg.count]

```
template<InputIterator Iter, class T>
  requires HasEqualTo<Iter::value_type, T>
  Iter::difference_type count(Iter first, Iter last, const T& value);

template<InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter::difference_type count_if(Iter first, Iter last, Pred pred);
```

1 *Effects:* Returns the number of iterators `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

2 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

25.1.10 Mismatch [mismatch]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2);

template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  pair<Iter1, Iter2> mismatch(Iter1 first1, Iter1 last1,
                              Iter2 first2, Pred pred);
```

1 *Returns:* A pair of iterators `i` and `j` such that `j == first2 + (i - first1)` and `i` is the first iterator in the range `[first1,last1)` for which the following corresponding conditions hold:

```
!(*i == *(first2 + (i - first1)))
pred(*i, *(first2 + (i - first1))) == false
```

Returns the pair `last1` and `first2 + (last1 - first1)` if such an iterator `i` is not found.

2 *Complexity:* At most `last1 - first1` applications of the corresponding predicate.

25.1.11 Equal [alg.equal]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2);

template<InputIterator Iter1, InputIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  bool equal(Iter1 first1, Iter1 last1,
             Iter2 first2, Pred pred);
```

1 *Returns:* `true` if for every iterator `i` in the range `[first1,last1)` the following corresponding conditions hold: `*i == *(first2 + (i - first1)), pred(*i, *(first2 + (i - first1))) != false`. Otherwise, returns `false`.

2 *Complexity:* At most `last1 - first1` applications of the corresponding predicate.

25.1.12 Search [alg.search]

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasEqualTo<Iter1::value_type, Iter2::value_type>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2);

template<ForwardIterator Iter1, ForwardIterator Iter2,
         Predicate<auto, Iter1::value_type, Iter2::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter1 search(Iter1 first1, Iter1 last1,
               Iter2 first2, Iter2 last2,
               Pred pred);
```

1 *Effects:* Finds a subsequence of equal values in a sequence.

2 *Returns:* The first iterator `i` in the range `[first1,last1 - (last2-first2))` such that for any non-negative integer `n` less than `last2 - first2` the following corresponding conditions hold: `*(i + n) == *(first2 + n), pred(*(i + n), *(first2 + n)) != false`. Returns `last1` if no such iterator is found.

3 *Complexity:* At most `(last1 - first1) * (last2 - first2)` applications of the corresponding predicate.

```
template<ForwardIterator Iter, class T>
  requires HasEqualTo<Iter::value_type, T>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value);

template<ForwardIterator Iter, class T,
         Predicate<auto, Iter::value_type, T> Pred>
  requires CopyConstructible<Pred>
  Iter search_n(Iter first, Iter last, Iter::difference_type count,
                const T& value, Pred pred);
```

4 *Effects:* Finds a subsequence of equal values in a sequence.

5 *Returns:* The first iterator `i` in the range `[first,last-count)` such that for any non-negative integer `n` less than `count` the following corresponding conditions hold: `*(i + n) == value, pred(*(i + n),value) != false`. Returns `last` if no such iterator is found.

6 *Complexity:* At most `last - first` applications of the corresponding predicate.

25.2 Mutating sequence operations [alg.modifying.operations]

25.2.1 Copy [alg.copy]

```
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy(InIter first, InIter last,
               OutIter result);
```

1 *Effects:* Copies elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last-first)`, performs `*(result + n) = *(first + n)`.

2 *Returns:* `result + (last - first)`.

Requires: `result` shall not be in the range `[first,last)`.

Complexity: Exactly `last - first` assignments.

```
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter copy_n(InIter first, InIter::difference_type n,
                 OutIter result);
```

Effects: For each non-negative integer `i < n`, performs `*(result + i) = *(first + i)`.

Returns: `result + n`.

Complexity: Exactly `n` assignments.

```
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter copy_if(InIter first, InIter last,
                  OutIter result, Pred pred);
```

Requires: The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

Effects: Copies all of the elements referred to by the iterator `i` in the range `[first,last)` for which `pred(*i)` is `true`.

Complexity: Exactly `last - first` applications of the corresponding predicate.

Remarks: Stable.

```
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires OutputIterator<OutIter, InIter::reference>
  OutIter copy_backward(InIter first, InIter last,
                        OutIter result);
```

Effects: Copies elements in the range `[first,last)` into the range `[result - (last-first),result)` starting from `last - 1` and proceeding to `first`.[264] For each positive integer `n <= (last - first)`, performs `*(result - n) = *(last - n)`.

Requires: `result` shall not be in the range `[first,last)`.

Returns: `result - (last - first)`.

Complexity: Exactly `last - first` assignments.

25.2.2 Move [alg.move]

```
template<InputIterator InIter, typename OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move(InIter first, InIter last,
               OutIter result);
```

Effects: Moves elements in the range `[first,last)` into the range `[result,result + (last - first))` starting from `first` and proceeding to `last`. For each non-negative integer `n < (last-first)`, performs `*(result + n) = std::move (*(first + n))`.

Returns: `result + (last - first)`.

264) `copy_backward` should be used instead of copy when `last` is in the range `[result - (last - first),result)`.

Requires: `result` shall not be in the range `[first,last)`.

Complexity: Exactly `last - first` move assignments.

```
template<BidirectionalIterator InIter, BidirectionalIterator OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::reference>::type>
  OutIter move_backward(InIter first, InIter last,
                        OutIter result);
```

Effects: Moves elements in the range `[first,last)` into the range `[result - (last-first),result)` starting from `last - 1` and proceeding to `first`.[265] For each positive integer `n <= (last - first)`, performs `*(result - n) = std::move(*(last - n))`.

Requires: `result` shall not be in the range `[first,last)`.

Returns: `result - (last - first)`.

Complexity: Exactly `last - first` assignments.

25.2.3 Swap [alg.swap]

```
template<class T>
  requires MoveAssignable<T> && MoveConstructible<T>
  void swap(T& a, T& b);
```

Effects: Exchanges values stored in two locations.

```
template<ValueType T, size_t N>
  requires Swappable<T>
  void swap(T (&a)[N], T (&b)[N]);
```

Effects: `swap_ranges(a, a + N, b)`

```
template<ForwardIterator Iter1, ForwardIterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  Iter2 swap_ranges(Iter1 first1, Iter1 last1,
                    Iter2 first2);
```

Effects: For each non-negative integer `n < (last1 - first1)` performs: `swap(*(first1 + n), *(first2 + n))`.

Requires: The two ranges `[first1,last1)` and `[first2,first2 + (last1 - first1))` shall not overlap.

Returns: `first2 + (last1 - first1)`.

Complexity: Exactly `last1 - first1` swaps.

```
template<Iterator Iter1, Iterator Iter2>
  requires HasSwap<Iter1::reference, Iter2::reference>
  void iter_swap(Iter1 a, Iter2 b);
```

Effects: `swap(*a, *b)`.

25.2.4 Transform [alg.transform]

265) `move_backward` should be used instead of move when `last` is in the range `[result - (last - first),result)`.

```
template<InputIterator InIter, class OutIter,
         Callable<auto, const InIter::value_type&> Op>
  requires OutputIterator<OutIter, Op::result_type>
        && CopyConstructible<Op>
  OutIter transform(InIter first, InIter last,
                    OutIter result, Op op);

template<InputIterator InIter1, InputIterator InIter2,
         class OutIter,
         Callable<auto, const InIter1::value_type&,
                        const InIter2::value_type&> BinaryOp>
  requires OutputIterator<OutIter, BinaryOp::result_type>
        && CopyConstructible<BinaryOp>
  OutIter transform(InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result,
                    BinaryOp binary_op);
```

1 *Effects:* Assigns through every iterator `i` in the range `[result,result + (last1 - first1))` a new corresponding value equal to `op(*(first1 + (i - result))` or `binary_op(*(first1 + (i - result), *(first2 + (i - result)))`.

2 *Requires:* `op` and `binary_op` shall not invalidate iterators or subranges, or modify elements in the ranges `[first1,last1]`, `[first2,first2 + (last1 - first1)]`, and `[result,result + (last1 - first1)]`.[266]

3 *Returns:* `result + (last1 - first1)`.

4 *Complexity:* Exactly `last1 - first1` applications of `op` or `binary_op`.

5 *Remarks:* `result` may be equal to `first` in case of unary transform, or to `first1` or `first2` in case of binary transform.

25.2.5 Replace [alg.replace]

```
template<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter, const T&>
        && HasEqualTo<Iter::value_type, T>
  void replace(Iter first, Iter last,
               const T& old_value, const T& new_value);

template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred, class T>
  requires OutputIterator<Iter, Iter::reference>
        && OutputIterator<Iter, const T&>
        && CopyConstructible<Pred>
  void replace_if(Iter first, Iter last,
                  Pred pred, const T& new_value);
```

1 *Requires:* The expression `*first = new_value` shall be valid.

2 *Effects:* Substitutes elements referred by the iterator `i` in the range `[first,last)` with `new_value`, when the following corresponding conditions hold: `*i == old_value`, `pred(*i) != false`.

3 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

266) The use of fully closed ranges is intentional.

```
template<InputIterator InIter, typename OutIter, class T>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const T&>
        && HasEqualTo<InIter::value_type, T>
  OutIter replace_copy(InIter first, InIter last,
                       OutIter result,
                       const T& old_value, const T& new_value);

template<InputIterator InIter, typename OutIter,
         Predicate<auto, InIter::value_type> Pred, class T>
  requires OutputIterator<OutIter, InIter::reference>
        && OutputIterator<OutIter, const T&>
        && CopyConstructible<Pred>
  OutIter replace_copy_if(InIter first, InIter last,
                          OutIter result,
                          Pred pred, const T& new_value);
```

4 *Requires:* The results of the expressions `*first` and `new_value` shall be writable to the `result` output iterator. The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

5 *Effects:* Assigns to every iterator `i` in the range `[result,result + (last - first))` either `new_value` or `*(first + (i - result))` depending on whether the following corresponding conditions hold:

```
*(first + (i - result)) == old_value
pred(*(first + (i - result))) != false
```

6 *Returns:* `result + (last - first)`.

7 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

25.2.6 Fill [alg.fill]

```
template<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, const T&>
  void fill(Iter first, Iter last, const T& value);

template<class Iter, IntegralLike Size, class T>
  requires OutputIterator<Iter, const T&>
  void fill_n(Iter first, Size n, const T& value);
```

1 *Effects:* The first algorithm assigns `value` through all the iterators in the range `[first,last)`. The second algorithm assigns `value` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing.

2 *Complexity:* Exactly `last - first`, `n`, or 0 assignments, respectively.

25.2.7 Generate [alg.generate]

```
template<ForwardIterator Iter, Callable Generator>
  requires OutputIterator<Iter, Generator::result_type>
        && CopyConstructible<Generator>
  void generate(Iter first, Iter last,
                Generator gen);

template<class Iter, IntegralLike Size, Callable Generator>
```

```
  requires OutputIterator<Iter, Generator::result_type>
        && CopyConstructible<Generator>
  void generate_n(Iter first, Size n, Generator gen);
```

1 *Effects:* The first algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,last)`. The second algorithm invokes the function object `gen` and assigns the return value of `gen` through all the iterators in the range `[first,first + n)` if `n` is positive, otherwise it does nothing.

2 *Complexity:* Exactly `last - first`, `n`, or 0 invocations of `gen` and assignments, respectively.

25.2.8 Remove [alg.remove]

```
template<ForwardIterator Iter, class T>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && HasEqualTo<Iter::value_type, T>
  Iter remove(Iter first, Iter last,
              const T& value);

template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && CopyConstructible<Pred>
  Iter remove_if(Iter first, Iter last,
                 Pred pred);
```

1 *Effects:* Eliminates all the elements referred to by iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == value, pred(*i) != false`.

2 *Returns:* The end of the resulting range.

3 *Remarks:* Stable.

4 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

```
template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter, class T>
  requires HasEqualTo<InIter::value_type, T>
  OutIter remove_copy(InIter first, InIter last,
                      OutIter result, const T& value);

template<InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter,
         Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  OutIter remove_copy_if(InIter first, InIter last,
                         OutIter result, Pred pred);
```

5 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

6 *Effects:* Copies all the elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions do not hold: `*i == value, pred(*i) != false`.

7 *Returns:* The end of the resulting range.

8 *Complexity:* Exactly `last - first` applications of the corresponding predicate.

9 *Remarks:* Stable.

25.2.9 Unique [alg.unique]

```
template<ForwardIterator Iter>
  requires OutputIterator<Iter, Iter::reference>
        && EqualityComparable<Iter::value_type>
  Iter unique(Iter first, Iter last);

template<ForwardIterator Iter, EquivalenceRelation<auto, Iter::value_type> Pred>
  requires OutputIterator<Iter, RvalueOf<Iter::reference>::type>
        && CopyConstructible<Pred>
  Iter unique(Iter first, Iter last,
              Pred pred);
```

1 *Effects:* For a nonempty range, eliminates all but the first element from every consecutive group of equivalent elements referred to by the iterator `i` in the range `[first + 1,last)` for which the following conditions hold: `*(i - 1) == *i` or `pred(*(i - 1), *i) != false`.

2 *Requires:* The comparison function shall be an equivalence relation.

3 *Returns:* The end of the resulting range.

4 *Complexity:* For nonempty ranges, exactly `(last - first) - 1` applications of the corresponding predicate.

```
template<InputIterator InIter, class OutIter>
  requires OutputIterator<OutIter, RvalueOf<InIter::value_type>::type>
      && EqualityComparable<InIter::value_type>
      && HasAssign<InIter::value_type, InIter::reference>
      && Constructible<InIter::value_type, InIter::reference>
  OutIter unique_copy(InIter first, InIter last, OutIter result);
template<InputIterator InIter, class OutIter,
         EquivalenceRelation<auto, InIter::value_type> Pred>
  requires OutputIterator<OutIter, RvalueOf<InIter::value_type>::type>
      && HasAssign<InIter::value_type, InIter::reference>
      && Constructible<InIter::value_type, InIter::reference>
      && CopyConstructible<Pred>
  OutIter unique_copy(InIter first, InIter last, OutIter result, Pred pred);
```

5 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

6 *Effects:* Copies only the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first,last)` for which the following corresponding conditions hold: `*i == *(i - 1)` or `pred(*i, *(i - 1)) != false`.

7 *Returns:* The end of the resulting range.

8 *Complexity:* For nonempty ranges, exactly `last - first - 1` applications of the corresponding predicate.

25.2.10 Reverse [alg.reverse]

```
template<BidirectionalIterator Iter>
  requires HasSwap<Iter::reference, Iter::reference>
  void reverse(Iter first, Iter last);
```

1 *Effects:* For each non-negative integer `i <= (last - first)/2`, applies `iter_swap` to all pairs of iterators `first + i, (last - i) - 1`.

2 *Complexity:* Exactly `(last - first)/2` swaps.

```
template<BidirectionalIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter reverse_copy(InIter first, InIter last, OutIter result);
```

3 *Effects:* Copies the range `[first,last)` to the range `[result,result+(last-first))` such that for any non-negative integer `i < (last - first)` the following assignment takes place: `*(result + (last - first) - i) = *(first + i)`.

4 *Requires:* The ranges `[first,last)` and `[result,result+(last-first))` shall not overlap.

5 *Returns:* `result + (last - first)`.

6 *Complexity:* Exactly `last - first` assignments.

25.2.11 Rotate [alg.rotate]

```
template<ShuffleIterator Iter>
  Iter rotate(Iter first, Iter middle,
              Iter last);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* `first + (last - middle)`.

3 *Remarks:* This is a left rotate.

4 *Requires:* `[first,middle)` and `[middle,last)` are valid ranges.

5 *Complexity:* At most `last - first` swaps.

```
template<ForwardIterator InIter, OutputIterator<auto, InIter::reference> OutIter>
  OutIter rotate_copy(InIter first, InIter middle,
                      InIter last, OutIter result);
```

6 *Effects:* Copies the range `[first,last)` to the range `[result,result + (last - first))` such that for each non-negative integer `i < (last - first)` the following assignment takes place: `*(result + i) = *(first + (i + (middle - first)) % (last - first))`.

7 *Returns:* `result + (last - first)`.

8 *Requires:* The ranges `[first,last)` and `[result,result + (last - first))` shall not overlap.

9 *Complexity:* Exactly `last - first` assignments.

25.2.12 Random shuffle [alg.random.shuffle]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
  void random_shuffle(Iter first,
                      Iter last);

template<RandomAccessIterator Iter, Callable<auto, Iter::difference_type> Rand>
  requires ShuffleIterator<Iter>
        && Convertible<Rand::result_type, Iter::difference_type>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand&& rand);
```

```
concept UniformRandomNumberGenerator<typename Rand> { }

template<RandomAccessIterator Iter, UniformRandomNumberGenerator Rand>
  void random_shuffle(Iter first,
                      Iter last,
                      Rand&& g);
```

1 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

2 The call `rand(n)` shall return a randomly chosen value in the interval `[0,n)`, for `n > 0`.

3 *Complexity:* Exactly `(last - first) - 1` swaps.

4 *Remarks:* To the extent that the implementation of these functions makes use of random numbers, the implementation shall use the following sources of randomness:

The underlying source of random numbers for the first form of the function is implementation-defined. An implementation may use the `rand` function from the standard C library.

In the second form of the function, the function object `rand` shall serve as the implementation's source of randomness.

In the third form of the function, the object `g` shall serve as the implementation's source of randomness.

25.2.13 Partitions [alg.partitions]

```
template <InputIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  bool is_partitioned(Iter first, Iter last, Pred pred);
```

1 *Returns:* `true` if `[first,last)` is partitioned by `pred`, i.e. if all elements that satisfy `pred` appear before those that do not.

2 *Complexity:* Linear. At most `last - first` applications of `pred`.

```
template<BidirectionalIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter partition(Iter first, Iter last, Pred pred);
```

3 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

4 *Returns:* An iterator `i` such that for any iterator `j` in the range `[first,i)` `pred(*j) != false`, and for any iterator `k` in the range `[i,last)`, `pred(*k) == false`.

5 *Complexity:* At most `(last - first)/2` swaps. Exactly `last - first` applications of the predicate are done.

```
template<BidirectionalIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Pred>
  Iter stable_partition(Iter first, Iter last, Pred pred);
```

6 *Effects:* Places all the elements in the range `[first,last)` that satisfy `pred` before all the elements that do not satisfy it.

7 *Returns:* An iterator `i` such that for any iterator `j` in the range `[first,i)`, `pred(*j) != false`, and for any iterator `k` in the range `[i,last)`, `pred(*k) == false`. The relative order of the elements in both groups is preserved.

8 *Complexity:* At most `(last - first) * log(last - first)` swaps, but only linear number of swaps if there is enough extra memory. Exactly `last - first` applications of the predicate.

```
template <InputIterator InIter, OutputIterator<auto, InIter::reference> OutIter1,
          OutputIterator<auto, InIter::reference> OutIter2, Predicate<auto, InIter::value_type> Pred>
  requires CopyConstructible<Pred>
  pair<OutIter1, OutIter2>
  partition_copy(InIter first, InIter last,
                 OutIter1 out_true, OutIter2 out_false,
                 Pred pred);
```

9 *Requires:* The input range shall not overlap with either of the output ranges.

10 *Effects:* For each iterator `i` in `[first,last)`, copies `*i` to the output range beginning with `out_true` if `pred(*i)` is `true`, or to the output range beginning with `out_false` otherwise.

11 *Returns:* A pair `p` such that `p.first` is the end of the output range beginning at `out_true` and `p.second` is the end of the output range beginning at `out_false`.

12 *Complexity:* Exactly `last - first` applications of `pred`.

```
template<ForwardIterator Iter, Predicate<auto, Iter::value_type> Pred>
  requires CopyConstructible<Pred>
  Iter partition_point(Iter first, Iter last, Pred pred);
```

13 *Returns:* An iterator `mid` such that `all_of(first, mid, pred)` and `none_of(mid, last, pred)` are both `true`.

14 *Complexity:* $\mathcal{O}((log(last - first)))$ applications of `pred`.

25.3 Sorting and related operations [alg.sorting]

1 All the operations in 25.3 have two versions: one that takes a function object of type `Compare` and one that uses an `operator<`.

2 `Compare` is used as a function object which returns `true` if the first argument is less than the second, and `false` otherwise. `Compare comp` is used throughout for algorithms assuming an ordering relation. It is assumed that `comp` will not apply any non-constant function through the dereferenced iterator.

3 For all algorithms that take `Compare`, there is a version that uses `operator<` instead. That is, `comp(*i, *j) != false` defaults to `*i < *j != false`. For algorithms other than those described in 25.3.3 to work correctly, `comp` has to induce a strict weak ordering on the values.

4 The term *strict* refers to the requirement of an irreflexive relation (`!comp(x, x)` for all `x`), and the term *weak* to requirements that are not as strong as those for a total ordering, but stronger than those for a partial ordering. If we define `equiv(a, b)` as `!comp(a, b) && !comp(b, a)`, then the requirements are that `comp` and `equiv` both be transitive relations:

- `comp(a, b) && comp(b, c)` implies `comp(a, c)`
- `equiv(a, b) && equiv(b, c)` implies `equiv(a, c)` [*Note:* Under these conditions, it can be shown that
  - `equiv` is an equivalence relation

— `comp` induces a well-defined relation on the equivalence classes determined by `equiv`

— The induced relation is a strict total ordering. — *end note*]

5 A sequence is *sorted with respect to a comparator* `comp` if for any iterator `i` pointing to the sequence and any non-negative integer `n` such that `i + n` is a valid iterator pointing to an element of the sequence, `comp(*(i + n), *i) == false`.

6 A sequence `[start,finish)` is *partitioned with respect to an expression* `f(e)` if there exists an integer `n` such that for all `0 <= i < distance(start, finish)`, `f(*(start + i))` is true if and only if `i < n`.

7 In the descriptions of the functions that deal with ordering relationships we frequently use a notion of equivalence to describe concepts such as stability. The equivalence to which we refer is not necessarily an `operator==`, but an equivalence relation induced by the strict weak ordering. That is, two elements `a` and `b` are considered equivalent if and only if `!(a < b) && !(b < a)`.

25.3.1 Sorting [alg.sort]

25.3.1.1 `sort` [sort]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
          StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort(Iter first, Iter last,
            Compare comp);
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Complexity:* $\mathcal{O}(N \log(N))$ (where N `== last - first`) comparisons.

25.3.1.2 `stable_sort` [stable.sort]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void stable_sort(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void stable_sort(Iter first, Iter last,
                   Compare comp);
```

1 *Effects:* Sorts the elements in the range `[first,last)`.

2 *Complexity:* It does at most $N \log^2(N)$ (where N `== last - first`) comparisons; if enough extra memory is available, it is $N \log(N)$.

3 *Remarks:* Stable.

25.3.1.3 partial_sort [partial.sort]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void partial_sort(Iter first,
                    Iter middle,
                    Iter last,
                    Compare comp);
```

1 *Effects:* Places the first `middle - first` sorted elements from the range `[first,last)` into the range `[first,middle)`. The rest of the elements in the range `[middle,last)` are placed in an unspecified order.

2 *Complexity:* It takes approximately `(last - first) * log(middle - first)` comparisons.

25.3.1.4 partial_sort_copy [partial.sort.copy]

```
template<InputIterator InIter, RandomAccessIterator RAIter>
  requires ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && HasLess<InIter::value_type, RAIter::value_type>
        && LessThanComparable<RAIter::value_type>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last);

template<InputIterator InIter, RandomAccessIterator RAIter, class Compare>
  requires ShuffleIterator<RAIter>
        && OutputIterator<RAIter, InIter::reference>
        && Predicate<Compare, InIter::value_type, RAIter::value_type>
        && StrictWeakOrder<Compare, RAIter::value_type>}
        && CopyConstructible<Compare>
  RAIter partial_sort_copy(InIter first, InIter last,
                           RAIter result_first, RAIter result_last,
                           Compare comp);
```

1 *Effects:* Places the first `min(last - first, result_last - result_first)` sorted elements into the range `[result_first,result_first + min(last - first, result_last - result_first))`.

2 *Returns:* The smaller of: `result_last` or `result_first + (last - first)`.

3 *Complexity:* Approximately `(last - first) * log(min(last - first, result_last - result_first))` comparisons.

25.3.1.5 is_sorted [is.sorted]

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_sorted(Iter first, Iter last);
```

1 *Returns:* `is_sorted_until(first, last) == last`

```
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  bool is_sorted(Iter first, Iter last,
                 Compare comp);
```

2 *Returns:* `is_sorted_until(first, last, comp) == last`

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter is_sorted_until(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter is_sorted_until(Iter first, Iter last,
                       Compare comp);
```

3 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is sorted.

4 *Complexity:* Linear.

25.3.2 Nth element [alg.nth.element]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void nth_element(Iter first, Iter nth,
                   Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void nth_element(Iter first, Iter nth,
                   Iter last,  Compare comp);
```

1 After `nth_element` the element in the position pointed to by `nth` is the element that would be in that position if the whole range were sorted. Also for any iterator `i` in the range `[first,nth)` and any iterator `j` in the range `[nth,last)` it holds that: `!(*i > *j)` or `comp(*j, *i) == false`.

2 *Complexity:* Linear on average.

25.3.3 Binary search [alg.binary.search]

1 All of the algorithms in this section are versions of binary search and assume that the sequence being searched is partitioned with respect to an expression formed by binding the search key to an argument of the implied or explicit comparison function. They work on non-random access iterators minimizing the number of comparisons, which will be logarithmic for all types of iterators. They are especially appropriate for random access iterators, because these algorithms do a logarithmic number of steps through the data structure. For non-random access iterators they execute a linear number of steps.

25.3.3.1 `lower_bound` [lower.bound]

```
template<ForwardIterator Iter, class T>
  requires HasLess<Iter::value_type, T>
  Iter lower_bound(Iter first, Iter last,
                   const T& value);

template<ForwardIterator Iter, class T, Predicate<auto, Iter::value_type, T> Compare>
  requires CopyConstructible<Compare>
  Iter lower_bound(Iter first, Iter last,
                   const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expression `e < value` or `comp(e, value)`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last]` such that for any iterator `j` in the range `[first,i)` the following corresponding conditions hold: `*j < value` or `comp(*j, value) != false`.

3 *Complexity:* At most $\texttt{log}_2\texttt{(last - first)} + \mathcal{O}(1)$ comparisons.

25.3.3.2 `upper_bound` [upper.bound]

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::value_type>
  Iter upper_bound(Iter first, Iter last,
                   const T& value);

template<ForwardIterator Iter, class T, Predicate<auto, T, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter upper_bound(Iter first, Iter last,
                   const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expression `!(value < e)` or `!comp(value, e)`.

2 *Returns:* The furthermost iterator `i` in the range `[first,last)` such that for any iterator `j` in the range `[first,i)` the following corresponding conditions hold: `!(value < *j)` or `comp(value, *j) == false`.

3 *Complexity:* At most $\texttt{log}_2\texttt{(last - first)} + \mathcal{O}(1)$ comparisons.

25.3.3.3 `equal_range` [equal.range]

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::value_type>
        && HasLess<Iter::value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value);

template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_type, T>
  pair<Iter, Iter>
    equal_range(Iter first,
                Iter last, const T& value,
                Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` implies `!(value < e)` or `comp(e, value)` shall implies `!comp(value, e)`.

2 *Returns:*

```
make_pair(lower_bound(first, last, value),
          upper_bound(first, last, value))
```

or

```
make_pair(lower_bound(first, last, value, comp),
          upper_bound(first, last, value, comp))
```

3 *Complexity:* At most `2 * log(last - first) + 1` comparisons.

25.3.3.4 `binary_search` [binary.search]

```
template<ForwardIterator Iter, class T>
  requires HasLess<T, Iter::value_type>
        && HasLess<Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value);

template<ForwardIterator Iter, class T, CopyConstructible Compare>
  requires Predicate<Compare, T, Iter::value_type>
        && Predicate<Compare, Iter::value_type, T>
  bool binary_search(Iter first, Iter last,
                     const T& value, Compare comp);
```

1 *Requires:* The elements `e` of `[first,last)` are partitioned with respect to the expressions `e < value` and `!(value < e)` or `comp(e, value)` and `!comp(value, e)`. Also, for all elements `e` of `[first, last)`, `e < value` implies `!(value < e)` or `comp(e, value)` implies `!comp(value, e)`.

2 *Returns:* `true` if there is an iterator `i` in the range `[first,last)` that satisfies the corresponding conditions: `!(*i < value) && !(value < *i)` or `comp(*i, value) == false && comp(value, *i) == false`.

3 *Complexity:* At most $\texttt{log}_2\texttt{(last - first)} + \mathcal{O}(1)$ comparisons.

25.3.4 Merge [alg.merge]

```
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         Predicate<auto, InIter2::value_type, InIter1::value_type> Compare>
  requires OutputIterator<OutIter, InIter1::reference>
```

```
      && OutputIterator<OutIter, InIter2::reference>
      && CopyConstructible<Compare>
  OutIter merge(InIter1 first1, InIter1 last1,
                InIter2 first2, InIter2 last2,
                OutIter result, Compare comp);
```

1 *Effects:* Merges two sorted ranges `[first1,last1)` and `[first2,last2)` into the range `[result, result + (last1 - first1) + (last2 - first2))`.

2 The resulting range shall not overlap with either of the original ranges. The list will be sorted in non-decreasing order according to the ordering defined by `comp`; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `*i < *(i - 1)` or `comp(*i, *(i - 1))` will be false.

3 *Returns:* `result + (last1 - first1) + (last2 - first2)`.

4 *Complexity:* At most `(last1 - first1) + (last2 - first2) - 1` comparisons.

5 *Remarks:* Stable.

```
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
      && LessThanComparable<Iter::value_type>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
      && CopyConstructible<Compare>
  void inplace_merge(Iter first,
                     Iter middle,
                     Iter last, Compare comp);
```

6 *Effects:* Merges two sorted consecutive ranges `[first,middle)` and `[middle,last)`, putting the result of the merge into the range `[first,last)`. The resulting range will be in non-decreasing order; that is, for every iterator `i` in `[first,last)` other than `first`, the condition `*i < *(i - 1)` or, respectively, `comp(*i, *(i - 1))` will be false.

7 *Complexity:* When enough additional memory is available, `(last - first) - 1` comparisons. If no additional memory is available, an algorithm with complexity $N \log(N)$ (where `N` is equal to `last - first`) may be used.

8 *Remarks:* Stable.

25.3.5 Set operations on sorted structures [alg.set.operations]

1 This section defines all the basic set operations on sorted structures. They also work with `multiset`s (23.3.4) containing multiple copies of equivalent elements. The semantics of the set operations are generalized to `multiset`s in a standard way by defining `set_union()` to contain the maximum number of occurrences of every element, `set_intersection()` to contain the minimum, and so on.

25.3.5.1 `includes` [includes]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasLess<Iter1::value_type, Iter2::value_type>
      && HasLess<Iter2::value_type, Iter1::value_type>
```

```
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2,
         typename Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::value_type>
  bool includes(Iter1 first1, Iter1 last1,
                Iter2 first2, Iter2 last2,
                Compare comp);
```

1 *Returns:* `true` if every element in the range `[first2, last2)` is contained in the range `[first1, last1)`. Returns `false` otherwise.

2 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

25.3.5.2 `set_union` [set.union]

```
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_union(InIter1 first1, InIter1 last1,
                    InIter2 first2, InIter2 last2,
                    OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_union(InIter1 first1, InIter1 last1,
                    InIter2 first2, InIter2 last2,
                    OutIter result, Compare comp);
```

1 *Effects:* Constructs a sorted union of the elements from the two ranges; that is, the set of elements that are present in one or both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1, last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then all m elements from the first range shall be copied to the output range, in order, and then $\max(n - m, 0)$ elements from the second range shall be copied to the output range, in order.

25.3.5.3 `set_intersection` [set.intersection]

```
template<InputIterator InIter1, InputIterator InIter2,
```

```
        typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_intersection(InIter1 first1, InIter1 last1,
                           InIter2 first2, InIter2 last2,
                           OutIter result, Compare comp);
```

1 *Effects:* Constructs a sorted intersection of the elements from the two ranges; that is, the set of elements that are present in both of the ranges.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the first $\min(m, n)$ elements shall be copied from the first range to the output range, in order.

25.3.5.4 `set_difference` [set.difference]

```
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter,
         CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_difference(InIter1 first1, InIter1 last1,
                         InIter2 first2, InIter2 last2,
                         OutIter result, Compare comp);
```

1 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, the last $\max(m-n,0)$ elements from `[first1,last1)` shall be copied to the output range.

25.3.5.5 `set_symmetric_difference` [set.symmetric.difference]

```
template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && HasLess<InIter2::value_type, InIter1::value_type>
        && HasLess<InIter1::value_type, InIter2::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                   InIter2 first2, InIter2 last2,
                                   OutIter result);

template<InputIterator InIter1, InputIterator InIter2,
         typename OutIter, CopyConstructible Compare>
  requires OutputIterator<OutIter, InIter1::reference>
        && OutputIterator<OutIter, InIter2::reference>
        && Predicate<Compare, InIter1::value_type, InIter2::value_type>
        && Predicate<Compare, InIter2::value_type, InIter1::value_type>
  OutIter set_symmetric_difference(InIter1 first1, InIter1 last1,
                                   InIter2 first2, InIter2 last2,
                                   OutIter result, Compare comp);
```

1 *Effects:* Copies the elements of the range `[first1,last1)` which are not present in the range `[first2, last2)`, and the elements of the range `[first2,last2)` which are not present in the range `[first1, last1)` to the range beginning at `result`. The elements in the constructed range are sorted.

2 *Requires:* The resulting range shall not overlap with either of the original ranges.

3 *Returns:* The end of the constructed range.

4 *Complexity:* At most `2 * ((last1 - first1) + (last2 - first2)) - 1` comparisons.

5 *Remarks:* If `[first1,last1)` contains m elements that are equivalent to each other and `[first2, last2)` contains n elements that are equivalent to them, then $|m-n|$ of those elements shall be copied to the output range: the last $m-n$ of these elements from `[first1,last1)` if $m > n$, and the last $n-m$ of these elements from `[first2,last2)` if $m < n$.

25.3.6 Heap operations [alg.heap.operations]

1 A *heap* is a particular organization of elements in a range between two random access iterators `[a,b)`. Its two key properties are:

(1) There is no element greater than `*a` in the range and

(2) `*a` may be removed by `pop_heap()`, or a new element added by `push_heap()`, in $\mathcal{O}(\log(N))$ time.

2 These properties make heaps useful as priority queues.

3 `make_heap()` converts a range into a heap and `sort_heap()` turns a heap into a sorted sequence.

25.3.6.1 `push_heap` [push.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter>
        && LessThanComparable<Iter::value_type>
  void push_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void push_heap(Iter first, Iter last,
                 Compare comp);
```

1 *Effects:* Places the value in the location `last - 1` into the resulting heap `[first,last)`.

2 *Requires:* The range `[first,last - 1)` shall be a valid heap.

3 *Complexity:* At most `log(last - first)` comparisons.

25.3.6.2 `pop_heap` [pop.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter> && LessThanComparable<Iter::value_type>
  void pop_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void pop_heap(Iter first, Iter last,
                Compare comp);
```

1 *Effects:* Swaps the value in the location `first` with the value in the location `last - 1` and makes `[first,last - 1)` into a heap.

2 *Requires:* The range `[first,last)` shall be a valid heap.

3 *Complexity:* At most `2 * log(last - first)` comparisons.

25.3.6.3 `make_heap` [make.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter> &&
           LessThanComparable<Iter::value_type>
  void make_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
```

```
      && CopyConstructible<Compare>
  void make_heap(Iter first, Iter last,
                 Compare comp);
```

1 *Effects:* Constructs a heap out of the range `[first,last)`.

2 *Complexity:* At most `3 * (last - first)` comparisons.

25.3.6.4 `sort_heap` [sort.heap]

```
template<RandomAccessIterator Iter>
  requires ShuffleIterator<Iter> && LessThanComparable<Iter::value_type>
  void sort_heap(Iter first, Iter last);

template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
        && CopyConstructible<Compare>
  void sort_heap(Iter first, Iter last,
                 Compare comp);
```

1 *Effects:* Sorts elements in the heap `[first,last)`.

2 *Requires:* The range `[first,last)` shall be a valid heap.

3 *Complexity:* At most $N \log(N)$ comparisons (where `N == last - first`).

25.3.6.5 `is_heap` [is.heap]

```
template<RandomAccessIterator Iter>
  requires LessThanComparable<Iter::value_type>
  bool is_heap(Iter first, Iter last);
```

1 *Returns:* `is_heap_until(first, last) == last`

```
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  bool is_heap(Iter first, Iter last, Compare comp);
```

2 *Returns:* `is_heap_until(first, last, comp) == last`

```
template<RandomAccessIterator Iter>
  Iter is_heap_until(Iter first, Iter last);
template<RandomAccessIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter is_heap_until(Iter first, Iter last,
                     Compare comp);
```

3 *Returns:* If `distance(first, last) < 2`, returns `last`. Otherwise, returns the last iterator `i` in `[first,last]` for which the range `[first,i)` is a heap.

4 *Complexity:* Linear.

25.3.7 Minimum and maximum [alg.min.max]

```
template<LessThanComparable T> const T& min(const T& a, const T& b);
template<class T, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& min(const T& a, const T& b, Compare comp);
```

1 *Returns:* The smaller value.

2 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template<class T>
T min(initializer_list<T> t);
```

3 *Requires:* `T` is `LessThanComparable` and `CopyConstructible`.

4 *Returns:* the smallest value in the `initializer_list`.

5 *Remarks:* returns the leftmost argument when several arguments are equivalent to the smallest.

```
template<class T, class Compare>
T min(initializer_list<T> t, Compare comp);
```

6 *Requires:* type `T` is LessThanComparable and CopyConstructible.

7 *Returns:* the smallest value in the `initializer_list`.

8 *Remarks:* returns the leftmost argument when several arguments are equivalent to the smallest.

```
template<LessThanComparable T> const T& max(const T& a, const T& b);
template<class T, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  const T& max(const T& a, const T& b, Compare comp);
```

9 *Returns:* The larger value.

10 *Remarks:* Returns the first argument when the arguments are equivalent.

```
template <class T>
T max(initializer_list<T> t);
```

11 *Requires:* `T` is `LessThanComparable` and `CopyConstructible`.

12 *Returns:* The largest value in the `initializer_list`.

13 *Remarks:* returns the leftmost argument when several arguments are equivalent to the largest.

```
template<class T, class Compare>
T max(initializer_list<T> t, Compare comp);
```

14 *Requires:* type `T` is LessThanComparable and CopyConstructible.

15 *Returns:* the largest value in the initializer_list.

16 *Remarks:* returns the leftmost argument when several arguments are equivalent to the largest.

```
template<LessThanComparable T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T, StrictWeakOrder<auto, T> Compare>
  requires !SameType<T, Compare> && CopyConstructible<Compare>
  pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

17 *Returns:* `pair<const T&, const T&>(b, a)` if `b` is smaller than `a`, and `pair<const T&, const T&>(a, b)` otherwise.

18 *Remarks:* Returns `<pair<const T&, const T&>(a, b)` when the arguments are equivalent.

19 *Complexity:* Exactly one comparison.

```
template<class T>
pair<const T&, const T&> minmax(initializer_list<T> t);
```

20 *Requires:* `T` is `LessThanComparable` and `CopyConstructible`.

21 *Returns:* `pair<const T&, const T&>(x, y)` where `x` is the smallest value and `y` the largest value in the `initializer_list`.

22 *Remarks:* `x` is the leftmost argument when several arguments are equivalent to the smallest. `y` is the rightmost argument when several arguments are equivalent to the largest.

23 *Complexity:* At most (3/2)`sizeof...(Args)` applications of the corresponding predicate.

```
template<class T, class Compare>
pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);
```

24 *Requires:* type `T` is LessThanComparable and CopyConstructible.

25 *Returns:* `pair<const T&, const T&>(x, y)` where `x` is the smallest value and `y` largest value in the `initializer_list`.

26 *Remarks:* `x` is the leftmost argument when several arguments are equivalent to the smallest. `y` is the rightmost argument when several arguments are equivalent to the largest.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter min_element(Iter first, Iter last);

template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter min_element(Iter first, Iter last,
                   Compare comp);
```

27 *Returns:* The first iterator `i` in the range `[first,last)` such that for any iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*j < *i)` or `comp(*j, *i) == false`. Returns `last` if `first == last`.

28 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

```
template<ForwardIterator Iter>
  requires LessThanComparable<Iter::value_type>
  Iter max_element(Iter first, Iter last);

template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  Iter max_element(Iter first, Iter last,
                   Compare comp);
```

29 *Returns:* The first iterator `i` in the range `[first,last)` such that for any iterator `j` in the range `[first,last)` the following corresponding conditions hold: `!(*i < *j)` or `comp(*i, *j) == false`. Returns `last` if `first == last`.

30 *Complexity:* Exactly `max((last - first) - 1, 0)` applications of the corresponding comparisons.

```
template<ForwardIterator Iter>
```

```
  requires LessThanComparable<Iter::value_type>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last);
template<ForwardIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires CopyConstructible<Compare>
  pair<Iter, Iter>
    minmax_element(Iter first, Iter last, Compare comp);
```

31 *Returns:* `make_pair(m, M)`, where `m` is the first iterator in `[first,last)` such that no iterator in the range refers to a smaller element, and `M` is the last iterator in `[first,last)` such that no iterator in the range refers to a larger element.

32 *Complexity:* At most $max(\lfloor \frac{3}{2}(N-1) \rfloor, 0)$ applications of the corresponding predicate, where N is `distance(first, last)`.

25.3.8 Lexicographical comparison [alg.lex.comparison]

```
template<InputIterator Iter1, InputIterator Iter2>
  requires HasLess<Iter1::value_type, Iter2::value_type>
        && HasLess<Iter2::value_type, Iter1::value_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2);

template<InputIterator Iter1, InputIterator Iter2, CopyConstructible Compare>
  requires Predicate<Compare, Iter1::value_type, Iter2::value_type>
        && Predicate<Compare, Iter2::value_type, Iter1::value_type>
  bool lexicographical_compare(Iter1 first1, Iter1 last1,
                               Iter2 first2, Iter2 last2,
                               Compare comp);
```

1 *Returns:* `true` if the sequence of elements defined by the range `[first1,last1)` is lexicographically less than the sequence of elements defined by the range `[first2,last2)`.

Returns `false` otherwise.

2 *Complexity:* At most `2*min((last1 - first1), (last2 - first2))` applications of the corresponding comparison.

3 *Remarks:* If two sequences have the same number of elements and their corresponding elements are equivalent, then neither sequence is lexicographically less than the other. If one sequence is a prefix of the other, then the shorter sequence is lexicographically less than the longer sequence. Otherwise, the lexicographical comparison of the sequences yields the same result as the comparison of the first corresponding pair of elements that are not equivalent.

```
for ( ; first1 != last1 && first2 != last2 ; ++first1, ++first2) {
  if (*first1 < *first2) return true;
  if (*first2 < *first1) return false;
}
return first1 == last1 && first2 != last2;
```

25.3.9 Permutation generators [alg.permutation.generators]

```
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
```

```
      && LessThanComparable<Iter::value_type>
  bool next_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
      && CopyConstructible<Compare>
  bool next_permutation(Iter first, Iter last, Compare comp);
```

1 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the next permutation. The next permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`. If such a permutation exists, it returns `true`. Otherwise, it transforms the sequence into the smallest permutation, that is, the ascendingly sorted one, and returns `false`.

2 *Complexity:* At most `(last - first)/2` swaps.

```
template<BidirectionalIterator Iter>
  requires ShuffleIterator<Iter>
      && LessThanComparable<Iter::value_type>
  bool prev_permutation(Iter first, Iter last);

template<BidirectionalIterator Iter,
         StrictWeakOrder<auto, Iter::value_type> Compare>
  requires ShuffleIterator<Iter>
      && CopyConstructible<Compare>
  bool prev_permutation(Iter first, Iter last, Compare comp);
```

3 *Effects:* Takes a sequence defined by the range `[first,last)` and transforms it into the previous permutation. The previous permutation is found by assuming that the set of all permutations is lexicographically sorted with respect to `operator<` or `comp`.

4 *Returns:* `true` if such a permutation exists. Otherwise, it transforms the sequence into the largest permutation, that is, the descendingly sorted one, and returns `false`.

5 *Complexity:* At most `(last - first)/2` swaps.

25.4 C library algorithms [alg.c.library]

1 Table 91 describes some of the contents of the header `<cstdlib>`.

Table 91 — Header `<cstdlib>` synopsis

Type	Name(s)
Type:	`size_t`
Functions:	`bsearch` `qsort`

2 The contents are the same as the Standard C library header `<stdlib.h>` with the following exceptions:

3 The function signature:

```
bsearch(const void *, const void *, size_t, size_t,
  int (*)(const void *, const void *));
```

is replaced by the two declarations:

```
extern "C" void *bsearch(const void *key, const void *base,
                         size_t nmemb, size_t size,
                         int (*compar)(const void *, const void *));
extern "C++" void *bsearch(const void *key, const void *base,
                           size_t nmemb, size_t size,
                           int (*compar)(const void *, const void *));
```

both of which have the same behavior as the original declaration.

4 The function signature:

```
qsort(void *, size_t, size_t,
  int (*)(const void *, const void *));
```

is replaced by the two declarations:

```
extern "C" void qsort(void* base, size_t nmemb, size_t size,
                      int (*compar)(const void*, const void*));
extern "C++" void qsort(void* base, size_t nmemb, size_t size,
                        int (*compar)(const void*, const void*));
```

both of which have the same behavior as the original declaration. The behavior is undefined unless the objects in the array pointed to by `base` are of trivial type.

[*Note:* Because the function argument `compar()` may throw an exception, `bsearch()` and `qsort()` are allowed to propagate the exception (17.6.5.10). — *end note*]

See also: ISO C 7.10.5.

26 Numerics library [numerics]

1 This Clause describes components that C++ programs may use to perform seminumerical operations.

2 The following subclauses describe components for complex number types, random number generation, numeric (n-at-a-time) arrays, generalized numeric algorithms, and facilities included from the ISO C library, as summarized in Table 92.

Table 92 — Numerics library summary

Subclause		Header(s)
26.1	Requirements	
26.3	Complex Numbers	`<complex>`
26.4	Random number generation	`<random>`
26.5	Numeric arrays	`<valarray>`
26.6	Generalized numeric operations	`<numeric>`

26.7	C library	`<cmath>`
		`<ctgmath>`
		`<tgmath.h>`
		`<cstdlib>`

26.1 Numeric type requirements [numeric.requirements]

1 The `complex` and `valarray` components are parameterized by the type of information they contain and manipulate. A C++ program shall instantiate these components only with a type `T` that satisfies the following requirements:[267]

- `T` is not an abstract class (it has no pure virtual member functions);
- `T` is not a reference type;
- `T` is not cv-qualified;
- If `T` is a class, it has a public default constructor;
- If `T` is a class, it has a public copy constructor with the signature `T::T(const T&)`
- If `T` is a class, it has a public destructor;
- If `T` is a class, it has a public assignment operator whose signature is either `T& T::operator=(const T&)` or `T& T::operator=(T)`
- If `T` is a class, its assignment operator, copy and default constructors, and destructor shall correspond to each other in the following sense: Initialization of raw storage using the default constructor, followed by assignment, is semantically equivalent to initialization of raw storage using the copy constructor. Destruction of an object, followed by initialization of its raw storage using the copy constructor, is semantically equivalent to assignment to the original object.

267) In other words, value types. These include arithmetic types, pointers, the library class `complex`, and instantiations of `valarray` for value types.

[*Note:* This rule states that there shall not be any subtle differences in the semantics of initialization versus assignment. This gives an implementation considerable flexibility in how arrays are initialized.

[*Example:* An implementation is allowed to initialize a `valarray` by allocating storage using the `new` operator (which implies a call to the default constructor for each element) and then assigning each element its value. Or the implementation can allocate raw storage and use the copy constructor to initialize each element. — *end example*]

If the distinction between initialization and assignment is important for a class, or if it fails to satisfy any of the other conditions listed above, the programmer should use `vector` (23.2.6) instead of `valarray` for that class; — *end note*]

— If `T` is a class, it does not overload unary `operator&`.

2 If any operation on `T` throws an exception the effects are undefined.

3 In addition, many member and related functions of `valarray<T>` can be successfully instantiated and will exhibit well-defined behavior if and only if `T` satisfies additional requirements specified for each such member or related function.

4 [*Example:* It is valid to instantiate `valarray<complex>`, but `operator>()` will not be successfully instantiated for `valarray<complex>` operands, since `complex` does not have any ordering operators. — *end example*]

26.2 The floating-point environment [cfenv]

26.2.1 Header <cfenv> synopsis [cfenv.syn]

```
namespace std {
  // types
  typedef object type  fenv_t;
  typedef integer type fexcept_t;

  // functions
  int feclearexcept(int except);
  int fegetexceptflag(fexcept_t *pflag, int except);
  int feraiseexcept(int except);
  int fesetexceptflag(const fexcept_t *pflag, int except);
  int fetestexcept(int except);

  int fegetround(void);
  int fesetround(int mode);

  int fegetenv(fenv_t *penv);
  int feholdexcept(fenv_t *penv);
  int fesetenv(const fenv_t *penv);
  int feupdateenv(const fenv_t *penv);
}
```

1 The header also defines the macros:

```
FE_ALL_EXCEPT
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
```

```
FE_UNDERFLOW

FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD

FE_DFL_ENV
```

2 The header defines all functions, types, and macros the same as C99 7.6.

26.2.2 Header <fenv.h> [fenv]

1 The header behaves as if it includes the header <cfenv>, and provides sufficient *using* declarations to declare in the global namespace all function and type names declared or defined in the header <cfenv>.

26.3 Complex numbers [complex.numbers]

1 The header <complex> defines a class template, and numerous functions for representing and manipulating complex numbers.

2 The effect of instantiating the template complex for any type other than float, double, or long double is unspecified. The specializations complex<float>, complex<double>, and complex<long double> are literal types (3.9).

3 If the result of a function is not mathematically defined or not in the range of representable values for its type, the behavior is undefined.

4 If z is an lvalue expression of type *cv* std::complex<T> then:

— the expression reinterpret_cast<cv T(&)[2]>(z) shall be well-formed,

— reinterpret_cast<cv T(&)[2]>(z)[0] shall designate the real part of z, and

— reinterpret_cast<cv T(&)[2]>(z)[1] shall designate the imaginary part of z.

Moreover, if a is an expression of type cv std::complex<T>* and the expression a[i] is well-defined for an integer expression i, then:

— reinterpret_cast<cv T*>(a)[2*i] shall designate the real part of a[i], and

— reinterpret_cast<cv T*>(a)[2*i + 1] shall designate the imaginary part of a[i].

26.3.1 Header <complex> synopsis [complex.synopsis]

```
namespace std {
  template<class T> class complex;
  template<> class complex<float>;
  template<> class complex<double>;
  template<> class complex<long double>;

  // 26.3.6 operators:
  template<class T>
    complex<T> operator+(const complex<T>&, const complex<T>&);
  template<class T> complex<T> operator+(const complex<T>&, const T&);
  template<class T> complex<T> operator+(const T&, const complex<T>&);

  template<class T> complex<T> operator-(
```

```
  const complex<T>&, const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&, const T&);
template<class T> complex<T> operator-(const T&, const complex<T>&);

template<class T> complex<T> operator*(
  const complex<T>&, const complex<T>&);
template<class T> complex<T> operator*(const complex<T>&, const T&);
template<class T> complex<T> operator*(const T&, const complex<T>&);

template<class T> complex<T> operator/(
  const complex<T>&, const complex<T>&);
template<class T> complex<T> operator/(const complex<T>&, const T&);
template<class T> complex<T> operator/(const T&, const complex<T>&);

template<class T> complex<T> operator+(const complex<T>&);
template<class T> complex<T> operator-(const complex<T>&);

template<class T> bool operator==(
  const complex<T>&, const complex<T>&);
template<class T> bool operator==(const complex<T>&, const T&);
template<class T> bool operator==(const T&, const complex<T>&);

template<class T> bool operator!=(const complex<T>&, const complex<T>&);
template<class T> bool operator!=(const complex<T>&, const T&);
template<class T> bool operator!=(const T&, const complex<T>&);

template<class T, class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>&, complex<T>&);

template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>&, const complex<T>&);

// 26.3.7 values:
template<class T> T real(const complex<T>&);
template<class T> T imag(const complex<T>&);

template<class T> T abs(const complex<T>&);
template<class T> T arg(const complex<T>&);
template<class T> T norm(const complex<T>&);

template<class T> complex<T> conj(const complex<T>&);
template <class T> complex<T> proj(const complex<T>&);
template<class T> complex<T> polar(const T&, const T& = 0);

// 26.3.8 transcendentals:
template<class T> complex<T> acos(const complex<T>&);
template<class T> complex<T> asin(const complex<T>&);
template<class T> complex<T> atan(const complex<T>&);

template<class T> complex<T> acosh(const complex<T>&);
template<class T> complex<T> asinh(const complex<T>&);
template<class T> complex<T> atanh(const complex<T>&);
```

```
  template<class T> complex<T> cos  (const complex<T>&);
  template<class T> complex<T> cosh (const complex<T>&);
  template<class T> complex<T> exp  (const complex<T>&);
  template<class T> complex<T> log  (const complex<T>&);
  template<class T> complex<T> log10(const complex<T>&);

  template<class T> complex<T> pow(const complex<T>&, const T&);
  template<class T> complex<T> pow(const complex<T>&, const complex<T>&);
  template<class T> complex<T> pow(const T&, const complex<T>&);

  template<class T> complex<T> sin  (const complex<T>&);
  template<class T> complex<T> sinh (const complex<T>&);
  template<class T> complex<T> sqrt (const complex<T>&);
  template<class T> complex<T> tan  (const complex<T>&);
  template<class T> complex<T> tanh (const complex<T>&);
}
```

26.3.2 Class template `complex` [complex]

```
namespace std {
  template<class T>
  class complex {
  public:
    typedef T value_type;

    complex(const T& re = T(), const T& im = T());
    complex(const complex&);
    template<class X> complex(const complex<X>&);

    T real() const;
    void real(T);
    T imag() const;
    void imag(T);

    complex<T>& operator= (const T&);
    complex<T>& operator+=(const T&);
    complex<T>& operator-=(const T&);
    complex<T>& operator*=(const T&);
    complex<T>& operator/=(const T&);

    complex& operator=(const complex&);
    template<class X> complex<T>& operator= (const complex<X>&);
    template<class X> complex<T>& operator+=(const complex<X>&);
    template<class X> complex<T>& operator-=(const complex<X>&);
    template<class X> complex<T>& operator*=(const complex<X>&);
    template<class X> complex<T>& operator/=(const complex<X>&);
  };
}
```

1 The class `complex` describes an object that can store the Cartesian components, `real()` and `imag()`, of a complex number.

26.3.3 `complex` specializations [complex.special]

```
namespace std {
```

```
template<> class complex<float> {
public:
  typedef float value_type;

  constexpr complex(float re = 0.0f, float im = 0.0f);
  explicit constexpr complex(const complex<double>&);
  explicit constexpr complex(const complex<long double>&);

  constexpr float real() const;
  void real(float);
  constexpr float imag() const;
  void imag(float);

  complex<float>& operator= (float);
  complex<float>& operator+=(float);
  complex<float>& operator-=(float);
  complex<float>& operator*=(float);
  complex<float>& operator/=(float);

  complex<float>& operator=(const complex<float>&);
  template<class X> complex<float>& operator= (const complex<X>&);
  template<class X> complex<float>& operator+=(const complex<X>&);
  template<class X> complex<float>& operator-=(const complex<X>&);
  template<class X> complex<float>& operator*=(const complex<X>&);
  template<class X> complex<float>& operator/=(const complex<X>&);
};

template<> class complex<double> {
public:
  typedef double value_type;

  constexpr complex(double re = 0.0, double im = 0.0);
  constexpr complex(const complex<float>&);
  explicit constexpr complex(const complex<long double>&);

  constexpr double real() const;
  void real(double);
  constexpr double imag() const;
  void imag(double);

  complex<double>& operator= (double);
  complex<double>& operator+=(double);
  complex<double>& operator-=(double);
  complex<double>& operator*=(double);
  complex<double>& operator/=(double);

  complex<double>& operator=(const complex<double>&);
  template<class X> complex<double>& operator= (const complex<X>&);
  template<class X> complex<double>& operator+=(const complex<X>&);
  template<class X> complex<double>& operator-=(const complex<X>&);
  template<class X> complex<double>& operator*=(const complex<X>&);
  template<class X> complex<double>& operator/=(const complex<X>&);
};

template<> class complex<long double> {
```

```
  public:
    typedef long double value_type;

    constexpr complex(long double re = 0.0L, long double im = 0.0L);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&);

    constexpr long double real() const;
    void real(long double);
    constexpr long double imag() const;
    void imag(long double);

    complex<long double>& operator=(const complex<long double>&);
    complex<long double>& operator= (long double);
    complex<long double>& operator+=(long double);
    complex<long double>& operator-=(long double);
    complex<long double>& operator*=(long double);
    complex<long double>& operator/=(long double);

    template<class X> complex<long double>& operator= (const complex<X>&);
    template<class X> complex<long double>& operator+=(const complex<X>&);
    template<class X> complex<long double>& operator-=(const complex<X>&);
    template<class X> complex<long double>& operator*=(const complex<X>&);
    template<class X> complex<long double>& operator/=(const complex<X>&);
  };
}
```

26.3.4 `complex` member functions [complex.members]

```
template<class T> complex(const T& re = T(), const T& im = T());
```

1 *Effects:* Constructs an object of class `complex`.

2 *Postcondition:* `real() == re && imag() == im`.

```
T real() const;
```

Returns: the value of the real component.

```
void real(T val);
```

Effects: Assigns `val` to the real component.

```
T imag() const;
```

Returns: the value of the imaginary component.

```
void imag(T val);
```

Effects: Assigns `val` to the imaginary component.

26.3.5 `complex` member operators [complex.member.ops]

```
complex<T>& operator+=(const T& rhs);
```

1 *Effects:* Adds the scalar value `rhs` to the real part of the complex value `*this` and stores the result in the real part of `*this`, leaving the imaginary part unchanged.

2 *Returns:* `*this`.

```
complex<T>& operator-=(const T& rhs);
```

3 *Effects:* Subtracts the scalar value `rhs` from the real part of the complex value `*this` and stores the result in the real part of `*this`, leaving the imaginary part unchanged.

4 *Returns:* `*this`.

```
complex<T>& operator*=(const T& rhs);
```

5 *Effects:* Multiplies the scalar value `rhs` by the complex value `*this` and stores the result in `*this`.

6 *Returns:* `*this`.

```
complex<T>& operator/=(const T& rhs);
```

7 *Effects:* Divides the scalar value `rhs` into the complex value `*this` and stores the result in `*this`.

8 *Returns:* `*this`.

```
complex<T>& operator+=(const complex<T>& rhs);
```

9 *Effects:* Adds the complex value `rhs` to the complex value `*this` and stores the sum in `*this`.

10 *Returns:* `*this`.

```
complex<T>& operator-=(const complex<T>& rhs);
```

11 *Effects:* Subtracts the complex value `rhs` from the complex value `*this` and stores the difference in `*this`.

12 *Returns:* `*this`.

```
complex<T>& operator*=(const complex<T>& rhs);
```

13 *Effects:* Multiplies the complex value `rhs` by the complex value `*this` and stores the product in `*this`.

Returns: `*this`.

```
complex<T>& operator/=(const complex<T>& rhs);
```

14 *Effects:* Divides the complex value `rhs` into the complex value `*this` and stores the quotient in `*this`.

15 *Returns:* `*this`.

26.3.6 `complex` non-member operations [complex.ops]

```
template<class T> complex<T> operator+(const complex<T>& lhs);
```

1 *Remarks:* unary operator.

2 *Returns:* `complex<T>(lhs)`.

```
template<class T>
  complex<T> operator+(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator+(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator+(const T& lhs, const complex<T>& rhs);
```

3 *Returns:* `complex<T>(lhs) += rhs`.

```
template<class T> complex<T> operator-(const complex<T>& lhs);
```

4 *Remarks:* unary operator.

5 *Returns:* `complex<T>(-lhs.real(),-lhs.imag())`.

```
template<class T>
  complex<T> operator-(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator-(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator-(const T& lhs, const complex<T>& rhs);
```

6 *Returns:* `complex<T>(lhs) -= rhs`.

```
template<class T>
  complex<T> operator*(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator*(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator*(const T& lhs, const complex<T>& rhs);
```

7 *Returns:* `complex<T>(lhs) *= rhs`.

```
template<class T>
  complex<T> operator/(const complex<T>& lhs, const complex<T>& rhs);
template<class T> complex<T> operator/(const complex<T>& lhs, const T& rhs);
template<class T> complex<T> operator/(const T& lhs, const complex<T>& rhs);
```

8 *Returns:* `complex<T>(lhs) /= rhs`.

```
template<class T>
  bool operator==(const complex<T>& lhs, const complex<T>& rhs);
template<class T> bool operator==(const complex<T>& lhs, const T& rhs);
template<class T> bool operator==(const T& lhs, const complex<T>& rhs);
```

9 *Returns:* `lhs.real() == rhs.real() && lhs.imag() == rhs.imag()`.

10 *Remarks:* The imaginary part is assumed to be `T()`, or 0.0, for the `T` arguments.

```
template<class T>
  bool operator!=(const complex<T>& lhs, const complex<T>& rhs);
template<class T> bool operator!=(const complex<T>& lhs, const T& rhs);
template<class T> bool operator!=(const T& lhs, const complex<T>& rhs);
```

11 *Returns:* `rhs.real() != lhs.real() || rhs.imag() != lhs.imag()`.

```
template<class T, class charT, class traits>
basic_istream<charT, traits>&
operator>>(basic_istream<charT, traits>& is, complex<T>& x);
```

12 *Effects:* Extracts a complex number `x` of the form: `u`, `(u)`, or `(u,v)`, where `u` is the real part and `v` is the imaginary part (27.6.1.2).

13 *Requires:* The input values shall be convertible to `T`.

If bad input is encountered, calls `is.setstate(ios_base::failbit)` (which may throw `ios::failure` (27.4.4.3).

14 *Returns:* `is`.

15 *Remarks:* This extraction is performed as a series of simpler extractions. Therefore, the skipping of whitespace is specified to be the same for each of the simpler extractions.

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x);
```

16 *Effects:* inserts the complex number `x` onto the stream `o` as if it were implemented as follows:

```
template<class T, class charT, class traits>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& o, const complex<T>& x) {
  basic_ostringstream<charT, traits> s;
  s.flags(o.flags());
  s.imbue(o.getloc());
  s.precision(o.precision());
  s << '(' << x.real() << "," << x.imag() << ')';
  return o << s.str();
}
```

17 *Note:* In a locale in which comma is used as a decimal point character, the use of comma as a field separator can be ambiguous. Inserting `std::ios_base::showpoint` into the output stream forces all outputs to show an explicit decimal point character; as a result, all inserted sequences of complex numbers can be extracted unambiguously.

26.3.7 `complex` value operations [complex.value.ops]

```
template<class T> T real(const complex<T>& x);
```

1 *Returns:* `x.real()`.

```
template<class T> T imag(const complex<T>& x);
```

2 *Returns:* `x.imag()`.

```
template<class T> T abs(const complex<T>& x);
```

3 *Returns:* the magnitude of `x`.

```
template<class T> T arg(const complex<T>& x);
```

4 *Returns:* the phase angle of `x`, or `atan2(imag(x), real(x))`.

```
template<class T> T norm(const complex<T>& x);
```

5 *Returns:* the squared magnitude of `x`.

```
template<class T> complex<T> conj(const complex<T>& x);
```

6 *Returns:* the complex conjugate of `x`.

```
template<class T> complex<T> proj(const complex<T>& x);
```

7 *Effects:* Behaves the same as the C99 function `cproj`, defined in 7.3.9.4.

```
template<class T> complex<T> polar(const T& rho, const T& theta = 0);
```

8 *Returns:* the `complex` value corresponding to a complex number whose magnitude is `rho` and whose phase angle is `theta`.

26.3.8 `complex` transcendentals [complex.transcendentals]

```
template<class T> complex<T> acos(const complex<T>& x);
```

1 *Effects:* Behaves the same as C99 function `cacos`, defined in 7.3.5.1.

```
template<class T> complex<T> asin(const complex<T>& x);
```

2 *Effects:* Behaves the same as C99 function `casin`, defined in 7.3.5.2.

```
template<class T> complex<T> atan(const complex<T>& x);
```

3 *Effects:* Behaves the same as C99 function `catan`, defined in 7.3.5.3.

```
template<class T> complex<T> acosh(const complex<T>& x);
```

4 *Effects:* Behaves the same as C99 function `cacosh`, defined in 7.3.6.1.

```
template<class T> complex<T> asinh(const complex<T>& x);
```

5 *Effects:* Behaves the same as C99 function `casinh`, defined in 7.3.6.2.

```
template<class T> complex<T> atanh(const complex<T>& x);
```

6 *Effects:* Behaves the same as C99 function `catanh`, defined in 7.3.6.3.

```
template<class T> complex<T> cos(const complex<T>& x);
```

7 *Returns:* the complex cosine of `x`.

```
template<class T> complex<T> cosh(const complex<T>& x);
```

8 *Returns:* the complex hyperbolic cosine of `x`.

```
template<class T> complex<T> exp(const complex<T>& x);
```

9 *Returns:* the complex base e exponential of `x`.

```
template<class T> complex<T> log(const complex<T>& x);
```

10 *Remarks:* the branch cuts are along the negative real axis.

11 *Returns:* the complex natural (base e) logarithm of `x`, in the range of a strip mathematically unbounded along the real axis and in the interval `[-i times pi,i times pi]` along the imaginary axis. When `x` is a negative real number, `imag(log(x))` is pi.

```
template<class T> complex<T> log10(const complex<T>& x);
```

12 *Remarks:* the branch cuts are along the negative real axis.

13 *Returns:* the complex common (base 10) logarithm of `x`, defined as `log(x)/log(10)`.

```
template<class T>
  complex<T> pow(const complex<T>& x, const complex<T>& y);
template<class T> complex<T> pow  (const complex<T>& x, const T& y);
template<class T> complex<T> pow  (const T& x, const complex<T>& y);
```

14 *Remarks:* the branch cuts are along the negative real axis.

15 *Returns:* the complex power of base `x` raised to the `y`-th power, defined as `exp(y*log(x))`. The value returned for `pow(0,0)` is implementation-defined.

```
template<class T> complex<T> sin  (const complex<T>& x);
```

16 *Returns:* the complex sine of `x`.

```
template<class T> complex<T> sinh (const complex<T>& x);
```

17 *Returns:* the complex hyperbolic sine of `x`.

```
template<class T> complex<T> sqrt (const complex<T>& x);
```

18 *Remarks:* the branch cuts are along the negative real axis.

19 *Returns:* the complex square root of `x`, in the range of the right half-plane. If the argument is a negative real number, the value returned lies on the positive imaginary axis.

```
template<class T> complex<T> tan  (const complex<T>& x);
```

20 *Returns:* the complex tangent of `x`.

```
template<class T> complex<T> tanh (const complex<T>& x);
```

21 *Returns:* the complex hyperbolic tangent of `x`.

26.3.9 Additional Overloads [cmplx.over]

1 The following function templates shall have additional overloads:

```
arg                 norm
conj                proj
imag                real
```

2 The additional overloads shall be sufficient to ensure:

1. If the argument has type `long double`, then it is effectively cast to `complex<long double>`.
2. Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.
3. Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

3 Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

26.3.10 Header <ccomplex> [ccmplx]

1 The header behaves as if it simply includes the header `<complex>`.

26.3.11 Header <complex.h> [cmplxh]

1 The header behaves as if it includes the header `<ccomplex>`. [*Note:* `<complex.h>` does not promote any interface into the global namespace as there is no C interface to promote. — *end note*]

26.4 Random number generation [rand]

1 This subclause defines a facility for generating (pseudo-)random numbers.

2 In addition to a few utilities, four categories of entities are described: *uniform random number generators*, *random number engines*, *random number engine adaptors*, and *random number distributions*. These categorizations are applicable to types that satisfy the corresponding requirements, to objects instantiated from

such types, and to templates producing such types when instantiated. [*Note:* These entities are specified in such a way as to permit the binding of any uniform random number generator object `e` as the argument to any random number distribution object `d`, thus producing a zero-argument function object such as given by `bind(d,e)`. — *end note*]

3 Each of the entities specified via this subclause has an associated arithmetic type (3.9.1) identified as `result_type`. With `T` as the `result_type` thus associated with such an entity, that entity is characterized

a) as *boolean* or equivalently as *boolean-valued*, if `T` is `bool`;

b) otherwise as *integral* or equivalently as *integer-valued*, if `numeric_limits<T>::is_integer` is `true`;

c) otherwise as *floating* or equivalently as *real-valued*.

If integer-valued, an entity may optionally be further characterized as *signed* or *unsigned*, according to `T`.

4 Unless otherwise specified, all descriptions of calculations in this subclause use mathematical real numbers.

5 Throughout this subclause, the operators bitand, bitor, and xor denote the respective conventional bitwise operations. Further,

a) the operator rshift denotes a bitwise right shift with zero-valued bits appearing in the high bits of the result, and

b) the operator lshift_w denotes a bitwise left shift with zero-valued bits appearing in the low bits of the result, and whose result is always taken modulo 2^w.

26.4.1 Requirements [rand.req]

26.4.1.1 General requirements [rand.req.genl]

1 Throughout this subclause 26.4, the effect of instantiating a template

a) that has a template type parameter named `UniformRandomNumberGenerator` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of uniform random number generator (26.4.1.2).

b) that has a template type parameter named `Engine` is undefined unless the corresponding template argument is cv-unqualified and satisfies the requirements of random number engine (26.4.1.3).

c) that has a template type parameter named `RealType` is undefined unless the corresponding template argument is cv-unqualified and is one of `float`, `double`, or `long double`.

d) that has a template type parameter named `IntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

e) that has a template type parameter named `UIntType` is undefined unless the corresponding template argument is cv-unqualified and is one of `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

2 All members declared `static const` in any of the following classes or class templates shall be defined in such a way that they are usable as integral constant expressions.

26.4.1.2 Uniform random number generator requirements [rand.req.urng]

1 A class `X` satisfies the requirements of a *uniform random number generator* if the expressions shown in table 93 are valid and have the indicated semantics. In that table,

a) `T` is the type named by `X`'s associated `result_type`, and

b) `u` is a value of `X`.

Table 93 — Uniform random number generator requirements

Expression	Return type	Pre-/post-condition	Complexity
`X::result_type`	`T`	`T` is an unsigned integer type (3.9.1).	compile-time
`u()`	`T`	Returns a value in the closed interval [`X::min()`, `X::max()`].	amortized constant
`X::min()`	`T`	Returns the least value potentially returned by `operator()`.	compile-time
`X::max()`	`T`	Returns the greatest value potentially returned by `operator()`.	compile-time

26.4.1.3 Random number engine requirements [rand.req.eng]

1 A class `X` that satisfies the requirements of a uniform random number generator (26.4.1.2) also satisfies the requirements of a *random number engine* if the expressions shown in table 94 are valid and have the indicated semantics, and if `X` also satisfies all other requirements of this section 26.4.1.3. In that table and throughout this section 26.4.1.3,

a) `T` is the type named by `X`'s associated `result_type`;

b) `u` is a value of `X`, `v` is an lvalue of `X`, `x` and `y` are (possibly `const`) values of `X`;

c) `s` is a value of arithmetic type (3.9.1);

d) `q` is an lvalue of type `seed_seq` (26.4.7.1);

e) `z` is a value of type `unsigned long long`;

f) `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and

g) `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to 21 and 27.

2 A random number engine object `x` has at any given time a state $\mathtt{x}_i$ for some integer $i \geq 0$. Upon construction, a random number engine `x` has an initial state $\mathtt{x}_0$. An engine's state may be established by invoking a constructor, `seed` member function, `operator=`, or a suitable `operator>>`.

3 The specification of each random number engine defines the size of its state in multiples of the size of its `result_type`, given as an integral constant expression. The specification of each random number engine also defines

a) the *transition algorithm* `TA` by which the engine's state $\mathtt{x}_i$ is advanced to its *successor state* $\mathtt{x}_{i+1}$, and

b) the *generation algorithm* GA by which an engine's state is mapped to a value of type result_type.

Table 94 — Random number engine requirements

Expression	Return type	Pre-/post-condition	Complexity
X()	—	Creates an engine with the same initial state as all other default-constructed engines of type X.	$\mathcal{O}(sizeof state)$
X(x)	—	Creates an engine that compares equal to x.	$\mathcal{O}(sizeof state)$
X(s)	—	Creates an engine with initial state determined by static_cast<X::result_type>(s).	$\mathcal{O}(sizeof state)$
X(q)[268]	—	Creates an engine u with an initial state that depends on a sequence produced by one call to q.generate.	Same as complexity of q.generate when called on a sequence whose length is size of state.
u.seed()	void	post: u == X()	same as X()
u.seed(s)	void	post: u == X(s)	same as X(s)
u.seed(q)	void	post: u == X(q).	same as X(q)
u()	T	Sets the state to $\mathsf{u}_{i+1} = \mathsf{TA}(\mathsf{u}_i)$ and returns $\mathsf{GA}(\mathsf{u}_i)$.	amortized constant
u.discard(z) [269]	void	post: The state of u is identical to that produced by z consecutive calls to u().	no worse than the complexity of z consecutive calls to u()
x == y	bool	With S_x and S_y as the infinite sequences of values that would be generated by repeated future calls to x() and y(), respectively, returns true if $S_x = S_y$; returns false otherwise.	$\mathcal{O}(sizeof state)$
x != y	bool	!(x == y)	$\mathcal{O}(sizeof state)$

268) This constructor (as well as the corresponding seed() function below) may be particularly useful to applications requiring a large number of independent random sequences.
269) This operation is common in user code, and can often be implemented in an engine-specific manner so as to provide significant performance improvements over an equivalent naive loop that makes z consecutive calls to u().

Table 94 — Random number engine requirements (continued)

Expression	Return type	Pre-/post-condition	Complexity
`os << x`	reference to the type of `os`	With `os.`*fmtflags* set to `ios_base::dec|ios_base::left` and the fill character set to the space character, writes to `os` the textual representation of `x`'s current state. In the output, adjacent numbers are separated by one or more space characters. post: The `os.`*fmtflags* and fill character are unchanged.	$\mathcal{O}(sizeof state)$
`is >> v`	reference to the type of `is`	With `is.fmtflags` set to `ios_base::dec`, sets `v`'s state as determined by reading its textual representation from `is`. If bad input is encountered, ensures that `v`'s state is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (27.4.4.3)). pre: The textual representation was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is.`*fmtflags* are unchanged.	$\mathcal{O}(sizeof state)$

4 `X` shall satisfy the requirements of uniform random number generator (26.4.1.2) as well as of `CopyConstructible` (20.1.8) and of `CopyAssignable` (20.1.8). Copy construction and assignment shall each be of complexity $\mathcal{O}(sizeof state)$.

5 If a textual representation written via `os << x` was subsequently read via `is >> v`, then `x == v` provided that there have been no intervening invocations of `x` or of `v`.

26.4.1.4 Random number engine adaptor requirements [rand.req.adapt]

1 A *random number engine adaptor* is a random number engine that takes values produced by some other random number engine or engines, and applies an algorithm to those values in order to deliver a sequence of values with different randomness properties. Engines adapted in this way are termed *base engines* in this context. The terms *unary*, *binary*, and so on, may be used to characterize an adaptor depending on the number n of base engines that adaptor utilizes.

2 A class X satisfies the requirements of a random number engine adaptor if the expressions shown in table 95 are valid and have the indicated semantics, and if X and its associated types also satisfies all other requirements of this section 26.4.1.4. In that table and throughout this section,

a) B_i is the type of the i^{th} of X's base engines, $1 \leq i \leq n$; and

b) b_i is a value of B_i.

If X is unary, i is omitted and understood to be 1.

Table 95 — Random number engine adaptor requirements

Expression	Return type	Pre-/post-condition	Complexity
X::base$i$_type	B_i	—	compile time
X::base$i$()	const B_i&	Returns a reference to b_i.	constant

3 X shall satisfy the requirements of random number engine (26.4.1.3), subject to the following:

a) The base engines of X are arranged in an arbitrary but fixed order, and that order is consistently used whenever functions are applied to those base engines in turn.

b) The complexity of each function is at most the sum of the complexities of the corresponding functions applied to each base engine.

c) The state of X includes the state of each of its base engines. The size of X's state is no less than the sum of the base engine sizes. Copying X's state (*e.g.*, during copy construction or copy assignment), includes copying, in turn, each base engine of X.

d) The textual representation of X includes, in turn, the textual representation of each of its base engines.

e) When X::X is invoked with no arguments, each of X's base engines is constructed, in turn, as if by its respective default constructor. When X::X is invoked with an X::result_type value s, each of X's base engines is constructed, in turn, with the next available value from the list $s+0, s+1, \ldots$. When X::X is invoked with an argument of type seed_seq, each of X's base engines is constructed, in turn, with that object as argument.

4 X shall have one additional constructor with n or more parameters such that the type of parameter i, $1 \leq i \leq n$, is const B_i& and such that all remaining parameters, if any, have default values. The constructor shall construct X, initializing each of its base engines, in turn, with a copy of the value of the corresponding argument.

26.4.1.5 Random number distribution requirements [rand.req.dist]

1 A class X satisfies the requirements of a *random number distribution* if the expressions shown in table 96 are valid and have the indicated semantics, and if X and its associated types also satisfies all other requirements of this section 26.4.1.5. In that table and throughout this section,

a) T is the type named by X's associated result_type;

b) P is the type named by X's associated param_type;

c) u is a value of X and x is a (possibly const) value of X;

d) glb and lub are values of T respectively corresponding to the greatest lower bound and the least upper bound on the values potentially returned by u's operator(), as determined by the current values of u's parameters;

e) `p` is a value of `P`;

f) `e` is an lvalue of an arbitrary type that satisfies the requirements of a uniform random number generator (26.4.1.2);

g) `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`; and

h) `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`;

where `charT` and `traits` are constrained according to 21 and 27.

2 The specification of each random number distribution identifies an associated mathematical *probability density function* $p(z)$ or an associated *discrete probability function* $P(z_i)$. Such functions are typically expressed using certain externally-supplied quantities known as the *parameters of the distribution* . Such distribution parameters are identified in this context by writing, for example, $p(z \mid a, b)$ or $P(z_i \mid a, b)$, to name specific parameters, or by writing, for example, $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$, to denote a distribution's parameters `p` taken as a whole.

Table 96 — Random number distribution requirements

Expression	Return type	Pre-/post-condition	Complexity
`X::result_type`	`T`	`T` is an arithmetic type.	compile-time
`X::param_type`	`P`	—	compile-time
`X(p)`	—	Creates a distribution whose behavior is indistinguishable from that of a distribution newly constructed directly from the values used to construct `p`.	same as `p`'s construction
`u.reset()`	`void`	Subsequent uses of `u` do not depend on values produced by `e` prior to invoking `reset`.	constant
`x.param()`	`P`	Returns a value `p` such that `X(p).param() == p`.	no worse than the complexity of `X(p)`
`u.param(p)`	`void`	post: `u.param() == p`.	no worse than the complexity of `X(p)`
`u(e)`	`T`	With `p` = `u.param()`, the sequence of numbers returned by successive invocations with the same object `e` is randomly distributed according to the associated $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$ function.	amortized constant number of invocations of `e`
`u(e,p)`	`T`	The sequence of numbers returned by successive invocations with the same objects `e` and `p` is randomly distributed according to the associated $p(z \mid \{\texttt{p}\})$ or $P(z_i \mid \{\texttt{p}\})$ function.	—
`x.min()`	`T`	Returns `glb`.	constant
`x.max()`	`T`	Returns `lub`.	constant

Table 96 — Random number distribution requirements (continued)

Expression	Return type	Pre-/post-condition	Complexity
`os << x`	reference to the type of `os`	Writes to `os` a textual representation for the parameters and the additional internal data of `x`. post: The `os.`*fmtflags* and fill character are unchanged.	—
`is >> u`	reference to the type of `is`	Restores from `is` the parameters and additional internal data of `u`. If bad input is encountered, ensures that `u` is unchanged by the operation and calls `is.setstate(ios::failbit)` (which may throw `ios::failure` (27.4.4.3)). pre: `is` provides a textual representation that was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is.`*fmtflags* are unchanged.	—

3 `X` shall satisfy the requirements of `CopyConstructible` (20.1.8) and `CopyAssignable` (20.1.8).

4 The sequence of numbers produced by repeated invocations of `x(e)` shall be independent of any invocation of `os << x` or of any `const` member function of `X` between any of the invocations `x(e)`.

5 If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(e)` shall produce the same sequence of numbers as would repeated invocations of `x(e)`.

6 It is unspecified whether `X::param_type` is declared as a (nested) `class` or via a `typedef`. In this subclause 26.4, declarations of `X::param_type` are in the form of `typedef`s only for convenience of exposition.

7 `P` shall satisfy the requirements of `CopyConstructible`, `CopyAssignable`, and `EqualityComparable` (20.1.5).

8 For each of the constructors of `X` taking arguments corresponding to parameters of the distribution, `P` shall have a corresponding constructor subject to the same requirements and taking arguments identical in number, type, and default values. Moreover, for each of the member functions of `X` that return values corresponding to parameters of the distribution, `P` shall have a corresponding member function with the identical name, type, and semantics.

9 `P` shall have a declaration of the form

```
typedef X distribution_type;
```

26.4.2 Header <random> synopsis [rand.synopsis]

```
namespace std {
  // 26.4.3.1 Class template linear_congruential_engine
  template <class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential_engine;

  // 26.4.3.2 Class template mersenne_twister_engine
  template <class UIntType, size_t w, size_t n, size_t m, size_t r,
            UIntType a, size_t u, UIntType d, size_t s,
            UIntType b, size_t t,
            UIntType c, size_t l, UIntType f>
    class mersenne_twister_engine;

  // 26.4.3.3 Class template subtract_with_carry_engine
  template <class UIntType, size_t w, size_t s, size_t r>
    class subtract_with_carry_engine;

  // 26.4.4.1 Class template discard_block_engine
  template <class Engine, size_t p, size_t r>
    class discard_block_engine;

  // 26.4.4.2 Class template independent_bits_engine
  template <class Engine, size_t w, class UIntType>
    class independent_bits_engine;

  // 26.4.4.3 Class template shuffle_order_engine
  template <class Engine, size_t k>
    class shuffle_order_engine;

  // 26.4.5 Engines and engine adaptors with predefined parameters
  typedef see below minstd_rand0;
  typedef see below minstd_rand;
  typedef see below mt19937_64;
  typedef see below ranlux24_base;
  typedef see below ranlux48_base;
  typedef see below ranlux24;
  typedef see below ranlux48;
  typedef see below knuth_b;
  typedef see below default_random_engine;

  // 26.4.6 Class random_device
  class random_device;

  // 26.4.7.1 Class seed_seq
  class seed_seq;

  // 26.4.7.2 Function template generate_canonical
  template<class RealType, size_t bits, class UniformRandomNumberGenerator>
    RealType generate_canonical(UniformRandomNumberGenerator& g);

  // 26.4.8.1.1 Class template uniform_int_distribution
  template <class IntType = int>
    class uniform_int_distribution;
```

```
// 26.4.8.1.2 Class template uniform_real_distribution
template <class RealType = double>
  class uniform_real_distribution;

// 26.4.8.2.1 Class bernoulli_distribution
class bernoulli_distribution;

// 26.4.8.2.2 Class template binomial_distribution
template <class IntType = int>
  class binomial_distribution;

// 26.4.8.2.3 Class template geometric_distribution
template <class IntType = int>
  class geometric_distribution;

// 26.4.8.2.4 Class template negative_binomial_distribution
template <class IntType = int>
  class negative_binomial_distribution;

// 26.4.8.3.1 Class template poisson_distribution
template <class IntType = int>
  class poisson_distribution;

// 26.4.8.3.2 Class template exponential_distribution
template <class RealType = double>
  class exponential_distribution;

// 26.4.8.3.3 Class template gamma_distribution
template <class RealType = double>
  class gamma_distribution;

// 26.4.8.3.4 Class template weibull_distribution
template <class RealType = double>
  class weibull_distribution;

// 26.4.8.3.5 Class template extreme_value_distribution
template <class RealType = double>
  class extreme_value_distribution;

// 26.4.8.4.1 Class template normal_distribution
template <class RealType = double>
  class normal_distribution;

// 26.4.8.4.2 Class template lognormal_distribution
template <class RealType = double>
  class lognormal_distribution;

// 26.4.8.4.3 Class template chi_squared_distribution
template <class RealType = double>
  class chi_squared_distribution;

// 26.4.8.4.4 Class template cauchy_distribution
template <class RealType = double>
  class cauchy_distribution;
```

```
  // 26.4.8.4.5 Class template fisher_f_distribution
  template <class RealType = double>
    class fisher_f_distribution;

  // 26.4.8.4.6 Class template student_t_distribution
  template <class RealType = double>
    class student_t_distribution;

  // 26.4.8.5.1 Class template discrete_distribution
  template <class IntType = int>
    class discrete_distribution;

  // 26.4.8.5.2 Class template piecewise_constant_distribution
  template <class RealType = double>
    class piecewise_constant_distribution;

  // 26.4.8.5.3 Class template general_pdf_distribution
  template <class RealType = double>
    class general_pdf_distribution;
}
```

26.4.3 Random number engine class templates [rand.eng]

1 Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

2 Except as required by table 94, no function described in this section 26.4.3 throws an exception.

3 The class templates specified in this section 26.4.3 satisfy the requirements of random number engine (26.4.1.3). Descriptions are provided here only for operations on the engines that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.3.1 Class template `linear_congruential_engine` [rand.eng.lcong]

1 A `linear_congruential_engine` random number engine produces unsigned integer random numbers. The state x_i of a `linear_congruential_engine` object `x` is of size 1 and consists of a single integer. The transition algorithm is a modular linear function of the form $\mathsf{TA}(\mathsf{x}_i) = (a \cdot \mathsf{x}_i + c) \bmod m$; the generation algorithm is $\mathsf{GA}(\mathsf{x}_i) = \mathsf{x}_{i+1}$.

```
namespace std {
  template <class UIntType, UIntType a, UIntType c, UIntType m>
  class linear_congruential_engine {
  public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static const result_type multiplier = a;
    static const result_type increment = c;
    static const result_type modulus = m;
    static constexpr result_type min() { return c == 0u ? 1u : 0u; }
    static constexpr result_type max() { return m - 1u; }
    static const result_type default_seed = 1u;

    // constructors and seeding functions
    explicit linear_congruential_engine(result_type s = default_seed);
```

```
    explicit linear_congruential_engine(seed_seq& q);
    void seed(result_type s = default_seed);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
  };
}
```

2 The template parameter `UIntType` shall denote an unsigned integral type large enough to store values as large as $m - 1$. If the template parameter`m` is 0, the modulus m used throughout this section 26.4.3.1 is `numeric_limits<result_type>::max()` plus 1. [*Note:* The result need not be representable as a value of type `result_type`. — *end note*] Otherwise, the following relations shall hold: `a` $<$ `m` and `c` $<$ `m`.

3 The textual representation consists of the value of $\mathtt{x}_i$.

```
explicit linear_congruential_engine(result_type s = default_seed);
```

4 *Effects:* Constructs a `linear_congruential_engine` object. If $c \bmod m$ is 0 and `s` mod m is 0, sets the engine's state to 1, otherwise sets the engine's state to `s` mod m.

```
explicit linear_congruential_engine(seed_seq& q);
```

5 *Effects:* Constructs a `linear_congruential_engine` object. With $k = \left\lceil \frac{\log_2 m}{32} \right\rceil$ and a an array (or equivalent) of length $k + 3$, invokes `q.generate(`$a + 0$`,` $a + k + 3$`)` and then computes $S = \left(\sum_{j=0}^{k-1} a_{j+3} \cdot 2^{32j} \right) \bmod m$. If $c \bmod m$ is 0 and S is 0, sets the engine's state to 1, else sets the engine's state to S.

26.4.3.2 Class template `mersenne_twister_engine` [rand.eng.mers]

1 A `mersenne_twister_engine` random number engine[270] produces unsigned integer random numbers in the closed interval $[0, 2^w - 1]$. The state $\mathtt{x}_i$ of a `mersenne_twister_engine` object `x` is of size n and consists of a sequence X of n values of the type delivered by `x`; all subscripts applied to X are to be taken modulo n.

2 The transition algorithm employs a twisted generalized feedback shift register defined by shift values n and m, a twist value r, and a conditional xor-mask a. To improve the uniformity of the result, the bits of the raw shift register are additionally *tempered* (*i.e.*, scrambled) according to a bit-scrambling matrix defined by values u, d, s, b, t, c, and ℓ.

The state transition is performed as follows:

a) Concatenate the upper $w - r$ bits of X_{i-n} with the lower r bits of X_{i+1-n} to obtain an unsigned integer value Y.

b) With $\alpha = a \cdot (Y \mathsf{\ bitand\ } 1)$, set X_i to X_{i+m-n} xor $(Y \mathsf{\ rshift\ } 1)$ xor α.

3 The generation algorithm determines the unsigned integer values z_1, z_2, z_3, z_4 as follows, then delivers z_4 as its result:

a) Let $z_1 = X_i \mathsf{\ xor\ } \big((X_i \mathsf{\ rshift\ } u) \mathsf{\ bitand\ } d\big)$.

b) Let $z_2 = z_1 \mathsf{\ xor\ } \big((z_1 \mathsf{\ lshift\ }_w s) \mathsf{\ bitand\ } b\big)$.

270) The name of this engine refers, in part, to a property of its period: For properly-selected values of the parameters, the period is closely related to a large Mersenne prime number.

c) Let $z_3 = z_2 \,\mathsf{xor}\, \big((z_2 \,\mathsf{lshift}_w\, t) \,\mathsf{bitand}\, c\big)$.

d) Let $z_4 = z_3 \,\mathsf{xor}\, (z_3 \,\mathsf{rshift}\, \ell)$.

```
namespace std {
  template <class UIntType, size_t w, size_t n, size_t m, size_t r,
            UIntType a, size_t u, size_t s,
            UIntType b, size_t t,
            UIntType c, size_t l>
  class mersenne_twister_engine {
  public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static const size_t word_size = w;
    static const size_t state_size = n;
    static const size_t shift_size = m;
    static const size_t mask_bits = r;
    static const UIntType xor_mask = a;
    static const size_t tempering_u = u;
    static const size_t tempering_d = d;
    static const size_t tempering_s = s;
    static const UIntType tempering_b = b;
    static const size_t tempering_t = t;
    static const UIntType tempering_c = c;
    static const size_t tempering_l = l;
    static const size_t initialization_multiplier = f;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w - 1; }
    static const result_type default_seed = 5489u;

    // constructors and seeding functions
    explicit mersenne_twister_engine(result_type value = default_seed);
    explicit mersenne_twister_engine(seed_seq& q);
    void seed(result_type value = default_seed);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
  };
}
```

4 The following relations shall hold: $1 \le \mathtt{m} \le \mathtt{n}$; $0 \le \mathtt{r}, \mathtt{u}, \mathtt{s}, \mathtt{t}, \mathtt{l} \le \mathtt{w} \le$ `numeric_limits<result_type>::digits`; $0 \le \mathtt{a}, \mathtt{b}, \mathtt{c} \le 2^{\mathtt{w}} - 1$.

5 The textual representation of $\mathtt{x}_i$ consists of the values of $X_{i-n}, \ldots, X_{i-1}$, in that order.

```
explicit mersenne_twister_engine(result_type value = default_seed);
```

6 *Effects:* Constructs a `mersenne_twister_engine` object. Sets X_{-n} to `value` mod 2^w. Then, iteratively for $i = 1-n, \ldots, -1$, sets X_i to

$$\big[f \cdot \big(X_{i-1} \,\mathsf{xor}\, (X_{i-1} \,\mathsf{rshift}\, (w-2))\big) + i \bmod n\big] \bmod 2^w \,.$$

7 *Complexity:* $\mathcal{O}(n)$.

```
explicit mersenne_twister_engine(seed_seq& q);
```

8 *Effects:* Constructs a `mersenne_twister_engine` object. With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $n \cdot k$, invokes `q.generate(`$a + 0,\ a + n \cdot k$`)` and then, iteratively for $i = -n, \ldots, -1$, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+n)+j} \cdot 2^{32j}\right) \bmod 2^w$. Finally, if the most significant $w - r$ bits of X_{-n} are zero, and if each of the other resulting X_i is 0, changes X_{-n} to 2^{w-1}.

26.4.3.3 Class template `subtract_with_carry_engine` [rand.eng.sub]

1 A `subtract_with_carry_engine` random number engine produces unsigned integer random numbers.

2 The state x_i of a `subtract_with_carry_engine` object `x` is of size $\mathcal{O}(r)$, and consists of a sequence X of r integer values $0 \le X_i < m = 2^w$; all subscripts applied to X are to be taken modulo r. The state x_i additionally consists of an integer c (known as the *carry*) whose value is either 0 or 1.

3 The state transition is performed as follows:

a) Let $Y = X_{i-s} - X_{i-r} - c$.

b) Set X_i to $y = Y \bmod m$. Set c to 1 if $Y < 0$, otherwise set c to 0.

[*Note:* This algorithm corresponds to a modular linear function of the form $\mathsf{TA}(\mathsf{x}_i) = (a \cdot \mathsf{x}_i) \bmod b$, where b is of the form $m^r - m^s + 1$ and $a = b - (b-1)/m$. — *end note*]

4 The generation algorithm is given by $\mathsf{GA}(\mathsf{x}_i) = y$, where y is the value produced as a result of advancing the engine's state as described above.

```
namespace std {
  template <class UIntType, size_t w, size_t s, size_t r>
  class subtract_with_carry_engine {
  public:
    // types
    typedef UIntType result_type;

    // engine characteristics
    static const size_t word_size = w;
    static const size_t short_lag = s;
    static const size_t long_lag = r;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return m - 1; }
    static const result_type default_seed = 19780503u;

    // constructors and seeding functions
    explicit subtract_with_carry_engine(result_type value = default_seed);
    explicit subtract_with_carry_engine(seed_seq& q);
    void seed(result_type value = default_seed);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);
  };
}
```

5 The following relations shall hold: $0 < \mathsf{s} < \mathsf{r}$, and $0 < \mathsf{w} \le$ `numeric_limits<result_type>::digits`.

6 The textual representation consists of the values of $X_{i-r}, \ldots, X_{i-1}$, in that order, followed by c.

```
explicit subtract_with_carry_engine(result_type value = default_seed);
```

7 *Effects:* Constructs a `subtract_with_carry_engine` object. Sets the values of X_{-r}, ..., X_{-1}, in that order, as required below. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

8 *Required behavior:* First construct `e`, a `linear_congruential_engine` object, as if by the following definition:

```
linear_congruential_engine<result_type
                           40014u,0u,2147483563u> e(value == 0u ? default_seed : value);
```

To set an X_k, use new values z_0, ..., z_{n-1} obtained from n successive invocations of `e` taken modulo 2^{32}. Set X_k to $\left(\sum_{j=0}^{n-1} z_j \cdot 2^{32j}\right) \bmod m$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

9 *Complexity:* Exactly $n \cdot$ `r` invocations of `e`.

```
explicit subtract_with_carry_engine(seed_seq& q);
```

10 *Effects:* Constructs a `subtract_with_carry_engine` object. With $k = \lceil w/32 \rceil$ and a an array (or equivalent) of length $r \cdot k$, invokes `q.generate(`a`+0, `$a+r\cdot k$`)` and then, iteratively for $i = -r$, ..., -1, sets X_i to $\left(\sum_{j=0}^{k-1} a_{k(i+r)+j} \cdot 2^{32j}\right) \bmod m$. If X_{-1} is then 0, sets c to 1; otherwise sets c to 0.

26.4.4 Random number engine adaptor class templates [rand.adapt]

1 Except where specified otherwise, the complexity of all functions specified in the following sections is constant.

2 Except as required by table 94, no function described in this section 26.4.4 throws an exception.

3 The class templates specified in this section 26.4.4 satisfy the requirements of random number engine adaptor (26.4.1.4). Descriptions are provided here only for operations on the engine adaptors that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

26.4.4.1 Class template `discard_block_engine` [rand.adapt.disc]

1 A `discard_block_engine` random number engine adaptor produces random numbers selected from those produced by some base engine e. The state $\mathtt{x}_i$ of a `discard_block_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e` and an additional integer n. The size of the state is the size of e's state plus 1.

2 The transition algorithm discards all but $r > 0$ values from each block of $p \geq r$ values delivered by e. The state transition is performed as follows: If $n \geq r$, advance the state of `e` from $\mathtt{e}_i$ to $\mathtt{e}_{i+p-r}$ and set n to 0. In any case, then increment n and advance `e`'s then-current state $\mathtt{e}_j$ to $\mathtt{e}_{j+1}$.

3 The generation algorithm yields the value returned by the last invocation of `e()` while advancing `e`'s state as described above.

```
namespace std {
  template <class Engine, size_t p, size_t r>
  class discard_block_engine {
  public:
    // types
    typedef Engine base_type;
    typedef typename base_type::result_type result_type;

    // engine characteristics
```

```
    static const size_t block_size = p;
    static const size_t used_block = r;
    static constexpr result_type min() { return base_type::min(); }
    static constexpr result_type max() { return base_type::max(); }

    // constructors and seeding functions
    discard_block_engine();
    explicit discard_block_engine(const base_type& urng);
    explicit discard_block_engine(result_type s);
    explicit discard_block_engine(seed_seq& q);
    void seed();
    void seed(result_type s);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const base_type& base() const;

  private:
    base_type e;    // exposition only
    size_t n;       // exposition only
  };
}
```

4 The following relations shall hold: $1 \leq \mathtt{r} \leq \mathtt{p}$.

5 The textual representation consists of the textual representation of `e` followed by the value of `n`.

6 In addition to its behavior pursuant to section 26.4.1.4, each constructor that is not a copy constructor sets `n` to 0.

26.4.4.2 Class template `independent_bits_engine` [rand.adapt.ibits]

1 An `independent_bits_engine` random number engine adaptor combines random numbers that are produced by some base engine e, so as to produce random numbers with a specified number of bits w. The state $\mathtt{x}_i$ of an `independent_bits_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e`; the size of the state is the size of e's state.

2 The transition and generation algorithms are described in terms of the following integral constants:

a) Let $R = \mathtt{e.max()} - \mathtt{e.min()} + 1$ and $m = \lfloor \log_2 R \rfloor$.

b) With n as determined below, let $w_0 = \lfloor w/n \rfloor$, $n_0 = n - w \bmod n$, $y_0 = 2^{w_0} \lfloor R/2^{w_0} \rfloor$, and $y_1 = 2^{w_0+1} \lfloor R/2^{w_0+1} \rfloor$.

c) Let $n = \lceil w/m \rceil$ if and only if the relation $R - y_0 \leq \lfloor y_0/n \rfloor$ holds as a result. Otherwise let $n = 1 + \lceil w/m \rceil$.

[*Note:* The relation $w = n_0 w_0 + (n - n_0)(w_0 + 1)$ always holds. — *end note*]

3 The transition algorithm is carried out by invoking `e()` as often as needed to obtain n_0 values less than y_0 +`e.min()` and $n - n_0$ values less than $y_1 +$ `e.min()`.

4 The generation algorithm uses the values produced while advancing the state as described above to yield a quantity S obtained as if by the following algorithm:

```
S = 0;
for (k = 0; k ≠ n0; k += 1)  {
  do u = e() - e.min(); while (u ≥ y0);
  S = 2^w0 · S + u mod 2^w0;
}
for (k = n0; k ≠ n; k += 1)  {
  do u = e() - e.min(); while (u ≥ y1);
  S = 2^(w0+1) · S + u mod 2^(w0+1);
}

namespace std {
  template <class Engine, size_t w, class UIntType>
  class independent_bits_engine {
  public:
    // types
    typedef Engine base_type;
    typedef UIntType result_type;

    // engine characteristics
    static const size_t word_size = w;
    static constexpr result_type min() { return 0; }
    static constexpr result_type max() { return 2^w − 1; }

    // constructors and seeding functions
    independent_bits_engine();
    explicit independent_bits_engine(const base_type& urng);
    explicit independent_bits_engine(result_type s);
    explicit independent_bits_engine(seed_seq& q);
    void seed();
    void seed(result_type s);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const base_type& base() const;

  private:
    base_type e;   // exposition only
  };
}
```

5 The following relations shall hold: $0 < w \leq$ `numeric_limits<result_type>::digits`.

6 The textual representation consists of the textual representation of `e`.

26.4.4.3 Class template `shuffle_order_engine` [rand.adapt.shuf]

1 A `shuffle_order_engine` random number engine adaptor produces the same random numbers that are produced by some base engine e, but delivers them in a different sequence. The state $\mathtt{x}_i$ of a `shuffle_order_engine` engine adaptor object `x` consists of the state $\mathtt{e}_i$ of its base engine `e`, an additional value Y of the type delivered by `e`, and an additional sequence V of k values also of the type delivered by `e`. The size of the state is the size of e's state plus $k + 1$.

2 The transition algorithm permutes the values produced by e. The state transition is performed as follows:

a) Calculate an integer j as $\left\lfloor \frac{k \cdot (Y - b_{\min})}{b_{\max} - b_{\min} + 1} \right\rfloor$.

b) Set Y to V_j and then set V_j to `b()`.

3 The generation algorithm yields the last value of `Y` produced while advancing `e`'s state as described above.

```
namespace std {
  template <class Engine, size_t k>
  class shuffle_order_engine {
  public:
    // types
    typedef Engine base_type;
    typedef typename base_type::result_type result_type;

    // engine characteristics
    static const size_t table_size = k;
    static constexpr result_type min() { return base_type::min(); }
    static constexpr result_type max() { return base_type::max(); }

    // constructors and seeding functions
    shuffle_order_engine();
    explicit shuffle_order_engine(const base_type& urng);
    explicit shuffle_order_engine(result_type s);
    explicit shuffle_order_engine(seed_seq& q);
    void seed();
    void seed(result_type s);
    void seed(seed_seq& q);

    // generating functions
    result_type operator()();
    void discard(unsigned long long z);

    // property functions
    const base_type& base() const;

  private:
    base_type e;        // exposition only
    result_type Y;      // exposition only
    result_type V[k];   // exposition only
  };
}
```

4 The following relation shall hold: $1 \leq$ `k`.

5 The textual representation consists of the textual representation of `e`, followed by the `k` values of V, followed by the value of Y.

6 In addition to its behavior pursuant to section 26.4.1.4, each constructor that is not a copy constructor initializes `V[0]`, ..., `V[k-1]` and Y, in that order, with values returned by successive invocations of `e()`.

26.4.5 Engines and engine adaptors with predefined parameters [rand.predef]

```
typedef linear_congruential_engine<uint_fast32_t, 16807, 0, 2147483647>
        minstd_rand0;
```

1 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `minstd_rand0` shall produce the value 1043618065.

```
typedef linear_congruential_engine<uint_fast32_t, 48271, 0, 2147483647>
        minstd_rand;
```

2 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `minstd_rand` shall produce the value 399268537.

```
typedef mersenne_twister_engine<uint_fast32_t,
        32,624,397,31,0x9908b0df,11,0xffffffff,7,0x9d2c5680,15,0xefc60000,18,1812433253>
        mt19937;
```

3 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `mt19937` shall produce the value 4123659995.

```
typedef mersenne_twister_engine<uint_fast64_t,
        64,312,156,31,0xb5026f5aa96619e9,29,
        0x5555555555555555,17,
        0x71d67fffeda60000,37,
        0xff7eee000000000,43,
        6364136223846793005>
        mt19937_64;
```

4 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `mt19937_64` shall produce the value 9981545732273789042.

```
typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
        ranlux24_base;
```

5 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux24_base` shall produce the value 7937952.

```
typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
        ranlux48_base;
```

6 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux48_base` shall produce the value 61839128582725.

```
typedef discard_block_engine<ranlux24_base, 223, 23>
        ranlux24;
```

7 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux24` shall produce the value 9901578.

```
typedef discard_block_engine<ranlux48_base, 389, 11>
        ranlux48
```

8 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `ranlux48` shall produce the value 249142670248501.

```
typedef shuffle_order_engine<minstd_rand0,256>
        knuth_b;
```

9 *Required behavior:* The 10000 th consecutive invocation of a default-constructed object of type `knuth_b` shall produce the value 1112339016.

```
typedef implementation-defined default_random_engine;
```

10 *Required behavior:* The named entity shall meet the requirements of a Random Number Engine (26.4.1.3).

11 The choice of engine type named by this `typedef` is implementation defined. [*Note:* The implementation may select this type on the basis of performance, size, quality, or any combination of such factors, so as to provide at least acceptable engine behavior for relatively casual, inexpert, and/or lightweight use. Because different implementations may select different underlying engine types, code that uses this `typedef` need not generate identical sequences across implementations. — *end note*]

26.4.6 Class `random_device` [rand.device]

1 A `random_device` uniform random number generator produces non-deterministic random numbers. It satisfies the requirements of uniform random number generator (26.4.1.2).

2 If implementation limitations prevent generating non-deterministic random numbers, the implementation may employ a random number engine.

```
namespace std {
  class random_device {
  public:
    // types
    typedef unsigned int result_type;

    // generator characteristics
    static constexpr result_type min() { return see below; }
    static constexpr result_type max() { return see below; }

    // constructors
    explicit random_device(const string& token = implementation-defined);

    // generating functions
    result_type operator()();

    // property functions
    double entropy() const;

    random_device(const random_device& ) = delete;
    void operator=(const random_device& ) = delete;
  };
}
```

3 The values of the `min` and `max` members are identical to the values returned by `numeric_limits<result_type>::min()` and `numeric_limits<result_type>::max()`, respectively.

```
explicit random_device(const string& token = implementation-defined);
```

4 *Effects:* Constructs a `random_device` non-deterministic uniform random number generator object. The semantics and default value of the `token` parameter are implementation-defined.[271]

5 *Throws:* A value of an implementation-defined type derived from `exception` if the `random_device` could not be initialized.

```
double entropy() const;
```

271) The parameter is intended to allow an implementation to differentiate between different sources of randomness.

6 *Returns:* If the implementation employs a random number engine, returns 0.0. Otherwise, returns an entropy estimate[272] for the random numbers returned by `operator()`, in the range `min()` to $\log_2(\texttt{max}() + 1)$.

7 *Throws:* Nothing.

```
result_type operator()();
```

8 *Returns:* A non-deterministic random value, uniformly distributed between `min()` and `max()`, inclusive. It is implementation-defined how these values are generated.

9 *Throws:* A value of an implementation-defined type derived from `exception` if a random number could not be obtained.

26.4.7 Utilities [rand.util]

26.4.7.1 Class `seed_seq` [rand.util.seedseq]

1 An object of type `seed_seq` consumes a sequence of integer-valued data and produces a fixed number of unsigned integer values, $0 \leq i < 2^{32}$, based on the consumed data. [*Note:* Such an object provides a mechanism to avoid replication of streams of random variates. This can be useful in applications requiring large numbers of random number engines. — *end note*]

2 In addition to the requirements set forth below, instances of `seed_seq` shall meet the requirements of `CopyConstructible` (20.1.8) and of `Assignable` (23.1).

```
namespace std {
  class seed_seq {
  public:
    // types
    typedef uint_least32_t result_type;

    // constructors
    seed_seq();
    template<class InputIterator>
        seed_seq(InputIterator begin, InputIterator end,
      size_t u = numeric_limits<typename iterator_traits<InputIterator>::value_type>::digits};

    // generating functions
    template<class RandomAccessIterator>
      void generate(RandomAccessIterator begin, RandomAccessIterator end) const;

    // property functions
    size_t size() const;
    template<class OutputIterator> void param(OutputIterator dest) const;

  private:
    vector<result_type> v;    // exposition only
  };
}
```

```
explicit seed_seq();
```

272) If a device has n states whose respective probabilities are $P_0, \ldots, P_{n-1}$, the device entropy S is defined as $S = -\sum_{i=0}^{n-1} P_i \cdot \log P_i$.

3 *Effects:* Constructs a `seed_seq` object as if by default-constructing its member `v`.

4 *Throws:* Nothing.

```
template<class InputIterator>
  seed_seq(InputIterator begin, InputIterator end,
  size_t u = numeric_limits<typename iterator_traits<InputIterator>::value_type>::digits>);
```

5 *Requires:* `InputIterator` shall satisfy the requirements of an input iterator (24.1.2) such that `iterator_traits<InputIterator>::value_type` shall denote an integral type.

6 *Effects:* Constructs a `seed_seq` object by rearranging some or all of the bits of the supplied sequence `[begin,end)` of w-bit quantities into 32-bit units, as if by the following:

First extract the rightmost `u` bits from each of the `n = end - begin` elements of the supplied sequence and concatenate all the extracted bits to initialize a single (possibly very large) unsigned binary number, $b = \sum_{i=0}^{n-1} (\texttt{begin[}i\texttt{]} \mod 2^u) \cdot 2^{w \cdot i}$ (in which the bits of each `begin[i]` are treated as denoting an unsigned quantity). Then carry out the following algorithm:

```
v.clear();
if (w < 32)
  v.push_back(n);
for( ; n > 0; --n)
  v.push_back(b mod 2^32), b /= 2^32;
```

```
template<class RandomAccessIterator>
void generate(RandomAccessIterator begin, RandomAccessIterator end) const;
```

7 *Requires:* `RandomAccessIterator` shall meet the requirements of a random access iterator (24.1.6) such that `iterator_traits<RandomAccessIterator>::value_type` shall denote an unsigned integral type capable of accommodating 32-bit quantities.

8 *Effects:* Does nothing if `begin == end`. Otherwise, with $s =$ `v.size()` and $n =$ `end` $-$ `begin`, fills the supplied range [`begin`, `end`) according to the following algorithm in which each operation is to be carried out modulo 2^{32}, each indexing operator applied to `begin` is to be taken modulo n, and $T(x)$ is defined as x xor (x rshift 27):

a) By way of initialization, set each element of the range to the value `0x8b8b8b8b`. Additionally, for use in subsequent steps, let $p = (\texttt{n} - t)/2$ and let $q = p + t$, where

$t = (\texttt{n} \geq 623)\ ?\ 11 : (\texttt{n} \geq 68)\ ?\ 7 : (\texttt{n} \geq 39)\ ?\ 5 : (\texttt{n} \geq 7)\ ?\ 3 : (\texttt{n} - 1)/2;$

b) With `m` as the larger of `s + 1` and `n`, transform the elements of the range: iteratively for `k` $= 0, \ldots,$ `m` $- 1$, calculate values

$$r_1 = 1664525 \cdot \texttt{T(begin[k]}\ \texttt{xor}\ \texttt{begin[k} + \texttt{p]}\ \texttt{xor}\ \texttt{begin[k} - 1\texttt{]})$$

$$r_2 = r_1 + \begin{cases} \texttt{s} & ,\texttt{k} = 0 \\ \texttt{k} \bmod \texttt{n} + \texttt{v[k} - 1\texttt{]} & ,0 < \texttt{k} <= \texttt{s} \\ \texttt{k} \bmod \texttt{n} & ,\texttt{s} < \texttt{k} \end{cases}$$

and, in order, increment `begin[k` $+ p$`]` by r_1, increment `begin[x` $+ q$`]` by r_2, and set `begin[k]` to r_2.

c) Transform the elements of the rangethree more times, beginning where the previous step ended: iteratively for `k` $=$ `m`$, \ldots,$ `m` $+$ `n` $- 1$, calculate values

$$r_3 = 1566083941 \cdot \texttt{T}(\texttt{begin[k]} + \texttt{begin[k+p]} + \texttt{begin[k-1]})$$
$$r_4 = r_3 - (\texttt{k} \bmod \texttt{n})$$

and, in order, update `begin[k+p]` by xoring it with r_4, update `begin[k+q]` by xoring it with r_3, and set `begin[k]` to r_4.

9 *Throws:* Nothing.

```
size_t size() const;
```

10 *Returns:* The number of 32-bit units that would be returned by a call to `param()`.

```
template<class OutputIterator> void param(OutputIterator dest) const;
```

11 *Requires:* `OutputIterator` shall satisfy the requirements of an output iterator (24.1.3) such that `iterator_traits<OutputIterator>::value_type` shall be assignable from `result_type`.

12 *Effects:* Copies the sequence of prepared 32-bit units to the given destination, as if by executing the following statement:

```
copy(v.begin(), v.end(), dest);
```

26.4.7.2 Function template `generate_canonical` [rand.util.canonical]

1 Each function instantiated from the template described in this section 26.4.7.2 maps the result of one or more invocations of a supplied uniform random number generator `g` to one member of the specified `RealType` such that, if the values g_i produced by `g` are uniformly distributed, the instantiation's results t_j, $0 \le t_j < 1$, are distributed as uniformly as possible as specified below.

2 [*Note:* Obtaining a value in this way can be a useful step in the process of transforming a value generated by a uniform random number generator into a value that can be delivered by a random number distribution. — *end note*]

```
template<class RealType, size_t bits, class UniformRandomNumberGenerator>
RealType generate_canonical(UniformRandomNumberGenerator& g);
```

3 *Complexity:* Exactly $k = \max(1, \lceil b/\log_2 R \rceil)$ invocations of `g`, where b[273] is the lesser of `numeric_limits<RealType>::digits` and `bits`, and R is the value of `g.max - g.min + 1`.

4 *Required behavior:* Invokes `g()` k times to obtain values $g_0, \ldots, g_{k-1}$, respectively. Calculates a quantity

$$S = \sum_{i=0}^{k-1} (g_i - \texttt{g.min}) \cdot R^i$$

using arithmetic of type `RealType`.

5 *Returns:* S/R^k.

6 *Throws:* What and when `g` throws.

273) b is introduced to avoid any attempt to produce more bits of randomness than can be held in `RealType`.

26.4.8 Random number distribution class templates [rand.dist]

1 The classes and class templates specified in this section 26.4.8 satisfy all the requirements of random number distribution (26.4.1.5). Descriptions are provided here only for operations on the distributions that are not described in those requirements or for operations where there is additional semantic information. Declarations for copy constructors, for copy assignment operators, and for equality and inequality operators are not shown in the synopses.

2 The algorithms for producing each of the specified distributions are implementation-defined.

3 The value of each probability density function $p(z)$ and of each discrete probability function $P(z_i)$ specified in this section is 0 everywhere outside its stated domain.

26.4.8.1 Uniform distributions [rand.dist.uni]

26.4.8.1.1 Class template `uniform_int_distribution` [rand.dist.uni.int]

1 A `uniform_int_distribution` random number distribution produces random integers i, $a \le i \le b$, distributed according to the constant discrete probability function

$$P(i \mid a, b) = 1/(b - a + 1) \ .$$

```
namespace std {
  template <class IntType = int>
  class uniform_int_distribution {
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
    explicit uniform_int_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit uniform_int_distribution(IntType a = 0, IntType b = numeric_limits<IntType>::max());
```

2 *Requires:* `a` $\le$ `b`.

3 *Effects:* Constructs a `uniform_int_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
result_type a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
result_type b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.4.8.1.2 Class template `uniform_real_distribution` [rand.dist.uni.real]

1 A `uniform_real_distribution` random number distribution produces random numbers x, $a \le x < b$, distributed according to the constant probability density function

$$p(x \mid a, b) = 1/(b - a) \ .$$

```
namespace std {
  template <class RealType = double>
  class uniform_real_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
    explicit uniform_real_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit uniform_real_distribution(RealType a = 0.0, RealType b = 1.0);
```

2 *Requires:* `a` $\le$ `b` and `b` $-$ `a` $\le$ `numeric_limits<RealType>::max()`.

3 *Effects:* Constructs a `uniform_real_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
result_type a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
result_type b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.4.8.2 Bernoulli distributions [rand.dist.bern]

26.4.8.2.1 Class `bernoulli_distribution` [rand.dist.bern.bernoulli]

1 A `bernoulli_distribution` random number distribution produces `bool` values b distributed according to the discrete probability function

$$P(b \mid p) = \begin{cases} p & if \quad b = \texttt{true} \\ 1 - p & if \quad b = \texttt{false} \end{cases} .$$

```
namespace std {
  class bernoulli_distribution {
  public:
    // types
    typedef bool result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit bernoulli_distribution(double p = 0.5);
    explicit bernoulli_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

`explicit bernoulli_distribution(double p = 0.5);`

2 *Requires:* $0 \leq \texttt{p} \leq 1$.

3 *Effects:* Constructs a `bernoulli_distribution` object; `p` corresponds to the parameter of the distribution.

`double p() const;`

4 *Returns:* The value of the `p` parameter with which the object was constructed.

26.4.8.2.2 Class template `binomial_distribution` [rand.dist.bern.bin]

1 A `binomial_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid t, p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i} .$$

```
namespace std {
  template <class IntType = int>
  class binomial_distribution {
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit binomial_distribution(IntType t = 1, double p = 0.5);
    explicit binomial_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    IntType t() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

`explicit binomial_distribution(IntType t = 1, double p = 0.5);`

2 *Requires:* $0 \le \mathtt{p} \le 1$ and $0 \le \mathtt{t}$.

3 *Effects:* Constructs a `binomial_distribution` object; `t` and `p` correspond to the respective parameters of the distribution.

`IntType t() const;`

4 *Returns:* The value of the `t` parameter with which the object was constructed.

`double p() const;`

5 *Returns:* The value of the `p` parameter with which the object was constructed.

26.4.8.2.3 Class template `geometric_distribution` [rand.dist.bern.geo]

1 A `geometric_distribution` random number distribution produces integer values $i \ge 0$ distributed according to the discrete probability function

$$P(i \mid p) = p \cdot (1-p)^i \ .$$

```
namespace std {
  template <class IntType = int>
  class geometric_distribution {
  public:
    // types
```

```
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit geometric_distribution(double p = 0.5);
    explicit geometric_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
    };
}
```

```
explicit geometric_distribution(double p = 0.5);
```

2 *Requires:* $0 < \mathtt{p} < 1$.

3 *Effects:* Constructs a `geometric_distribution` object; `p` corresponds to the parameter of the distribution.

```
double p() const;
```

4 *Returns:* The value of the `p` parameter with which the object was constructed.

26.4.8.2.4 Class template `negative_binomial_distribution` [rand.dist.bern.negbin]

1 A `negative_binomial_distribution` random number distribution produces random integers $i \geq 0$ distributed according to the discrete probability function

$$P(i \mid k, p) = \binom{k+i-1}{i} \cdot p^k \cdot (1-p)^i \, .$$

```
namespace std {
  template <class IntType = int>
  class negative_binomial_distribution {
  public:
    // types
    typedef IntType  result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();
```

```
    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    IntType k() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit negative_binomial_distribution(IntType k = 1, double p = 0.5);
```

2 *Requires:* $0 < \mathsf{p} \leq 1$ and $0 < \mathsf{k}$.

3 *Effects:* Constructs a `negative_binomial_distribution` object; `k` and `p` correspond to the respective parameters of the distribution.

```
IntType k() const;
```

4 *Returns:* The value of the `k` parameter with which the object was constructed.

```
double p() const;
```

5 *Returns:* The value of the `p` parameter with which the object was constructed.

26.4.8.3 Poisson distributions [rand.dist.pois]

26.4.8.3.1 Class template `poisson_distribution` [rand.dist.pois.poisson]

1 A `poisson_distribution` random number distribution produces integer values $i \geq 0$ distributed according to the discrete probability function

$$P(i \,|\, \mu) = \frac{e^{-\mu}\mu^i}{i\,!} \ .$$

The distribution parameter μ is also known as this distribution's *mean* .

```
namespace std {
  template <class IntType = int>
  class poisson_distribution {
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit poisson_distribution(double mean = 1.0);
    explicit poisson_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
```

```
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    double mean() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit poisson_distribution(double mean = 1.0);
```

2 *Requires:* $0 <$ `mean`.

3 *Effects:* Constructs a `poisson_distribution` object; `mean` corresponds to the parameter of the distribution.

```
double mean() const;
```

4 *Returns:* The value of the `mean` parameter with which the object was constructed.

26.4.8.3.2 Class template `exponential_distribution` [rand.dist.pois.exp]

1 An `exponential_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \lambda) = \lambda e^{-\lambda x} \ .$$

```
namespace std {
  template <class RealType = double>
  class exponential_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit exponential_distribution(RealType lambda = 1.0);
    explicit exponential_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType lambda() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
```

```
    result_type max() const;
  };
}
```

```
explicit exponential_distribution(RealType lambda = 1.0);
```

2 *Requires:* $0 <$ `lambda`.

3 *Effects:* Constructs a `exponential_distribution` object; `lambda` corresponds to the parameter of the distribution.

```
RealType lambda() const;
```

4 *Returns:* The value of the `lambda` parameter with which the object was constructed.

26.4.8.3.3 Class template `gamma_distribution` [rand.dist.pois.gamma]

1 A `gamma_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid \alpha, \beta) = \frac{e^{-x/\beta}}{\beta^{\alpha} \cdot \Gamma(\alpha)} \cdot x^{\alpha-1} .$$

```
namespace std {
  template <class RealType = double>
  class gamma_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
    explicit gamma_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType alpha() const;
    RealType beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit gamma_distribution(RealType alpha = 1.0, RealType beta = 1.0);
```

2 *Requires:* $0 <$ `alpha` and $0 <$ `beta`.

3 *Effects:* Constructs a `gamma_distribution` object; `alpha` and `beta` correspond to the parameters of the distribution.

```
RealType alpha() const;
```

4 *Returns:* The value of the `alpha` parameter with which the object was constructed.

```
RealType beta() const;
```

5 *Returns:* The value of the `beta` parameter with which the object was constructed.

26.4.8.3.4 Class template `weibull_distribution` [rand.dist.pois.weibull]

1 A `weibull_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \mid a, b) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{a-1} \cdot \exp\left(-\left(\frac{x}{b}\right)^{a}\right) .$$

```
namespace std {
  template <class RealType = double>
  class weibull_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0)
    explicit weibull_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit weibull_distribution(RealType a = 1.0, RealType b = 1.0);
```

2 *Requires:* $0 <$ `a` and $0 <$ `b`.

3 *Effects:* Constructs a `weibull_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the a parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the b parameter with which the object was constructed.

26.4.8.3.5 Class template extreme_value_distribution [rand.dist.pois.extreme]

1 An extreme_value_distribution random number distribution produces random numbers x distributed according to the probability density function[274]

$$p(x \mid a, b) = \frac{1}{b} \cdot \exp\left(\frac{a-x}{b} - \exp\left(\frac{a-x}{b}\right)\right) .$$

```
namespace std {
  template <class RealType = double>
  class extreme_value_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
    explicit extreme_value_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit extreme_value_distribution(RealType a = 0.0, RealType b = 1.0);
```

2 *Requires:* $0 <$ b.

3 *Effects:* Constructs an extreme_value_distribution object; a and b correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the a parameter with which the object was constructed.

274) The distribution corresponding to this probability density function is also known (with a possible change of variable) as the Gumbel Type I, the log-Weibull, or the Fisher-Tippett Type I distribution.

```
RealType b() const;
```

Returns: The value of the `b` parameter with which the object was constructed.

26.4.8.4 Normal distributions [rand.dist.norm]

26.4.8.4.1 Class template `normal_distribution` [rand.dist.norm.normal]

A `normal_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid \mu, \sigma)p(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The distribution parameters μ and σ are also known as this distribution's *mean* and *standard deviation* .

```
namespace std {
  template <class RealType = double>
  class normal_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructors and reset functions
    explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
    explicit normal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType mean() const;
    RealType stddev() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit normal_distribution(RealType mean = 0.0, RealType stddev = 1.0);
```

Requires: $0 <$ `stddev`.

Effects: Constructs a `normal_distribution` object; `mean` and `stddev` correspond to the respective parameters of the distribution.

```
RealType mean() const;
```

Returns: The value of the `mean` parameter with which the object was constructed.

```
RealType stddev() const;
```

5 *Returns:* The value of the `stddev` parameter with which the object was constructed.

26.4.8.4.2 Class template `lognormal_distribution` [rand.dist.norm.lognormal]

1 A `lognormal_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid m, s) = \frac{1}{sx\sqrt{2\pi}} \cdot \exp\left(-\frac{(\ln x - m)^2}{2s^2}\right).$$

```
namespace std {
  template <class RealType = double>
  class lognormal_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType m() const;
    RealType s() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

`explicit lognormal_distribution(RealType m = 0.0, RealType s = 1.0);`

2 *Requires:* $0 <$ `s`.

3 *Effects:* Constructs a `lognormal_distribution` object; `m` and `s` correspond to the respective parameters of the distribution.

`RealType m() const;`

4 *Returns:* The value of the `m` parameter with which the object was constructed.

`RealType s() const;`

5 *Returns:* The value of the `s` parameter with which the object was constructed.

26.4.8.4.3 Class template `chi_squared_distribution` [rand.dist.norm.chisq]

1 A `chi_squared_distribution` random number distribution produces random numbers $x > 0$ distributed according to the probability density function

$$p(x \mid n) = \frac{x^{(n/2)-1} \cdot e^{-x/2}}{\Gamma(n/2) \cdot 2^{n/2}}$$

```
namespace std {
  template <class RealType = double>
  class chi_squared_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit chi_squared_distribution(RealType n = 1);
    explicit chi_squared_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

`explicit chi_squared_distribution(RealType n = 1);`

2 *Requires:* $0 <$ `n`.

3 *Effects:* Constructs a `chi_squared_distribution` object; `n` corresponds to the parameter of the distribution.

`RealType n() const;`

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.4.8.4.4 Class template `cauchy_distribution` [rand.dist.norm.cauchy]

1 A `cauchy_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid a, b) = \left(\pi b \left(1 + \left(\frac{x-a}{b} \right)^2 \right) \right)^{-1} .$$

```
namespace std {
  template <class RealType = double>
  class cauchy_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
    explicit cauchy_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit cauchy_distribution(RealType a = 0.0, RealType b = 1.0);
```

2 *Requires:* $0 < \texttt{b}$.

3 *Effects:* Constructs a `cauchy_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

```
RealType a() const;
```

4 *Returns:* The value of the `a` parameter with which the object was constructed.

```
RealType b() const;
```

5 *Returns:* The value of the `b` parameter with which the object was constructed.

26.4.8.4.5 Class template `fisher_f_distribution` [rand.dist.norm.f]

1 A `fisher_f_distribution` random number distribution produces random numbers $x \geq 0$ distributed according to the probability density function

$$p(x \,|\, m, n) = \frac{\Gamma\big((m+n)/2\big)}{\Gamma(m/2)\,\Gamma(n/2)} \cdot \Big(\frac{m}{n}\Big)^{m/2} \cdot x^{(m/2)-1} \cdot \Big(1 + \frac{mx}{n}\Big)^{-(m+n)/2}$$

```
namespace std {
  template <class RealType = double>
  class fisher_f_distribution {
  public:
```

```
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType m() const;
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit fisher_f_distribution(RealType m = 1, RealType n = 1);
```

2 *Requires:* $0 < \texttt{m}$ and $0 < \texttt{n}$.

3 *Effects:* Constructs a `fisher_f_distribution` object; `m` and `n` correspond to the respective parameters of the distribution.

```
RealType m() const;
```

4 *Returns:* The value of the `m` parameter with which the object was constructed.

```
RealType n() const;
```

5 *Returns:* The value of the `n` parameter with which the object was constructed.

26.4.8.4.6 Class template `student_t_distribution` [rand.dist.norm.t]

1 A `student_t_distribution` random number distribution produces random numbers x distributed according to the probability density function

$$p(x \mid n) = \frac{1}{\sqrt{n\pi}} \cdot \frac{\Gamma\big((n+1)/2\big)}{\Gamma(n/2)} \cdot \left(1 + \frac{x^2}{n}\right)^{-(n+1)/2}$$

```
namespace std {
  template <class RealType = double>
  class student_t_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;
```

```
    // constructor and reset functions
    explicit student_t_distribution(RealType n = 1);
    explicit student_t_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
explicit student_t_distribution(RealType n = 1);
```

2 *Requires:* $0 < \mathtt{n}$.

3 *Effects:* Constructs a `student_t_distribution` object; `n` and `n` correspond to the respective parameters of the distribution.

```
RealType n() const;
```

4 *Returns:* The value of the `n` parameter with which the object was constructed.

26.4.8.5 Sampling distributions [rand.dist.samp]

26.4.8.5.1 Class template `discrete_distribution` [rand.dist.samp.discrete]

1 A `discrete_distribution` random number distribution produces random integers i, $0 \le i < n$, distributed according to the discrete probability function

$$P(i \mid p_0, \ldots, p_{n-1}) = p_i \ .$$

```
namespace std {
  template <class IntType = int>
  class discrete_distribution {
  public:
    // types
    typedef IntType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    discrete_distribution();
    template <class InputIterator>
      discrete_distribution(InputIterator firstW, InputIterator lastW);
    explicit discrete_distribution(const param_type& parm);
    void reset();
```

```
    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    vector<double> probabilities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

`discrete_distribution();`

2 *Effects:* Constructs a `discrete_distribution` object with $n = 1$ and $p_0 = 1$. [*Note:* Such an object will always deliver the value 0. — *end note*]

```
template <class InputIterator>
discrete_distribution(InputIterator firstW, InputIterator lastW);
```

3 *Requires:*

a) `InputIterator` shall satisfy the requirements of an input iterator (24.1.2).

b) If `firstW == lastW`, let the sequence w have length $n = 1$ and consist of the single value $w_0 = 1$. Otherwise, `[firstW,lastW)` shall form a sequence w of length $n > 0$ and `*firstW` shall yield a value w_0 convertible to `double`. [*Note:* The values w_k are commonly known as the *weights* . — *end note*]

c) The following relations shall hold: $w_k \geq 0$ for $k = 0, \ldots, n-1$, and $0 < S = w_0 + \cdots + w_{n-1}$.

4 *Effects:* Constructs a `discrete_distribution` object with probabilities

$$p_k = \frac{w_k}{S} for k = 0, \ldots, n-1.$$

`vector<double> probabilities() const;`

5 *Returns:* A `vector<double>` whose `size` member returns n and whose `operator[]` member returns p_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.4.8.5.2 Class template `piecewise_constant_distribution` [rand.dist.samp.pconst]

1 A `piecewise_constant_distribution` random number distribution produces random numbers x, $b_0 \leq x < b_n$, uniformly distributed over each subinterval $[b_i, b_{i+1})$ according to the probability density function

$$p(x \,|\, b_0, \ldots, b_n,\ \rho_0, \ldots, \rho_{n-1}) = \rho_i \ , for b_i \leq x < b_{i+1} \ .$$

The $n + 1$ distribution parameters b_i are also known as this distribution's *interval boundaries*.

```
namespace std {
  template <class RealType = double>
  class piecewise_constant_distribution {
  public:
```

```
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    piecewise_constant_distribution();
    template <class InputIteratorB, class InputIteratorW>
      piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB,
                                      InputIteratorW firstW);
    explicit piecewise_constant_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    vector<RealType> intervals() const;
    vector<double> densities() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

`piecewise_constant_distribution();`

2 *Effects:* Constructs a `piecewise_constant_distribution` object with $n = 1$, $\rho_0 = 1$, $b_0 = 0$, and $b_1 = 1$.

```
template <class InputIteratorB, class InputIteratorW>
piecewise_constant_distribution(InputIteratorB firstB, InputIteratorB lastB, InputIteratorW firstW);
```

3 *Requires:*

a) `InputIteratorB` shall satisfy the requirements of an input iterator (24.1.2), as shall `InputIteratorW`.

b) If `firstB == lastB` or the sequence `w` has the length zero,

a) let the sequence w have length $n = 1$ and consist of the single value $w_0 = 1$, and

b) let the sequence b have length $n + 1$ with $b_0 = 0$ and $b_1 = 1$.

Otherwise,

c) `[firstB, lastB)` shall form a sequence b of length $n + 1$ whose leading element b_0 shall be convertible to `result_type`, and

d) the length of the sequence w starting from `firstW` shall be at least n, `*firstW` shall return a value w_0 that is convertible to `double`, and any w_k for $k \geq n$ shall be ignored by the distribution.

[*Note:* The values w_k are commonly known as the *weights* . — *end note*]

c) The following relations shall hold for $k = 0, \ldots, n - 1$: $b_k < b_{k+1}$ and $0 \leq w_k$. Also, $0 < S = w_0 + \cdots + w_{n-1}$.

4 *Effects:* Constructs a `piecewise_constant_distribution` object with probability densities

$$\rho_k = \frac{w_k}{S \cdot (b_{k+1} - b_k)} \; for k = 0, \ldots, n-1.$$

```
vector<result_type> intervals() const;
```

5 *Returns:* A `vector<result_type>` whose `size` member returns $n+1$ and whose `operator[]` member returns b_k when invoked with argument k for $k = 0, \ldots, n$.

```
vector<double> densities() const;
```

6 *Returns:* A `vector<result_type>` whose `size` member returns n and whose `operator[]` member returns ρ_k when invoked with argument k for $k = 0, \ldots, n-1$.

26.4.8.5.3 Class template `general_pdf_distribution` [rand.dist.samp.genpdf]

1 A `general_pdf_distribution` random number distribution produces random numbers x, $x_{\min} \le x < x_{\max}$, distributed according to the probability density function

$$p(x \,|\, x_{\min}, x_{\max}, \rho) = \rho(x) \; , for \; x_{\min} \le x < x_{\max} \; .$$

```
namespace std {
  template <class RealType = double>
  class general_pdf_distribution {
  public:
    // types
    typedef RealType result_type;
    typedef unspecified param_type;

    // constructor and reset functions
    general_pdf_distribution();
    template <class Func>
      general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);
    explicit general_pdf_distribution(const param_type& parm);
    void reset();

    // generating functions
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng);
    template <class UniformRandomNumberGenerator>
      result_type operator()(UniformRandomNumberGenerator& urng, const param_type& parm);

    // property functions
    result_type xmin() const;
    result_type xmax() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
}
```

```
general_pdf_distribution();
```

2 *Effects:* Constructs a `general_pdf_distribution` object with $x_{min} = 0$ and $x_{max} = 1$ such that $p(x) = 1$ for all $x_{min} \leq x < x_{max}$.

```
template <class Func>
general_pdf_distribution(result_type xmin, result_type xmax, Func pdf);
```

3 *Requires:*

a) `pdf` shall be callable with one argument of type `result_type`, and shall return values of a type convertible to `double`;

b) $x_{\min} < x_{\max}$, and for all $x_{\min} \leq x < x_{\max}$, `pdf(`$x$`)` shall return a value that is non-negative, non-NaN, and non-infinity; and

c) the following relations shall hold:

$$0 < z = \int_{x_{\min}}^{x_{\max}} f(x)\,dx < \infty\ ,$$

where f is the mathematical function corresponding to the supplied `pdf`. [*Note:* This implies that the user-supplied `pdf` need not be normalized. — *end note*]

4 *Effects:* Constructs a `general_pdf_distribution` object; `xmin` and `xmax` correspond to the respective parameters of the distribution and the corresponding probability density function is given by $\rho(x) = f(x)/z$.

```
result_type xmin() const;
```

5 *Returns:* The value of the `xmin` parameter with which the object was constructed.

```
result_type xmax() const;
```

6 *Returns:* The value of the `xmax` parameter with which the object was constructed.

26.5 Numeric arrays [numarray]

26.5.1 Header <valarray> synopsis [valarray.synopsis]

```
namespace std {
  template<class T> class valarray;         // An array of type T
  class slice;                              // a BLAS-like slice out of an array
  template<class T> class slice_array;
  class gslice;                             // a generalized slice out of an array
  template<class T> class gslice_array;
  template<class T> class mask_array;       // a masked array
  template<class T> class indirect_array;   // an indirected array

  template<class T> void swap(valarray<T>&, valarray<T>&);
  template<class T> void swap(valarray<T>&&, valarray<T>&);
  template<class T> void swap(valarray<T>&, valarray<T>&&);

  template<class T> valarray<T> operator* (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator* (const valarray<T>&, const T&);
  template<class T> valarray<T> operator* (const T&, const valarray<T>&);

  template<class T> valarray<T> operator/ (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
```

```
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);

template<class T> valarray<T> operator% (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);

template<class T> valarray<T> operator+ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);

template<class T> valarray<T> operator- (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);

template<class T> valarray<T> operator^ (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);

template<class T> valarray<T> operator& (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);

template<class T> valarray<T> operator| (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);

template<class T> valarray<T> operator<<(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const T&, const valarray<T>&);

template<class T> valarray<T> operator>>(const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);

template<class T> valarray<bool> operator&&(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T> valarray<bool> operator&&(const T&, const valarray<T>&);

template<class T> valarray<bool> operator||(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const T&);
template<class T> valarray<bool> operator||(const T&, const valarray<T>&);

template<class T>
  valarray<bool> operator==(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);
template<class T>
  valarray<bool> operator!=(const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);

template<class T>
  valarray<bool> operator< (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
template<class T> valarray<bool> operator< (const T&, const valarray<T>&);
```

```
  template<class T>
    valarray<bool> operator> (const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
  template<class T> valarray<bool> operator> (const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator<=(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
  template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);
  template<class T>
    valarray<bool> operator>=(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
  template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);

  template<class T> valarray<T> abs  (const valarray<T>&);
  template<class T> valarray<T> acos (const valarray<T>&);
  template<class T> valarray<T> asin (const valarray<T>&);
  template<class T> valarray<T> atan (const valarray<T>&);

  template<class T> valarray<T> atan2(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> atan2(const valarray<T>&, const T&);
  template<class T> valarray<T> atan2(const T&, const valarray<T>&);

  template<class T> valarray<T> cos  (const valarray<T>&);
  template<class T> valarray<T> cosh (const valarray<T>&);
  template<class T> valarray<T> exp  (const valarray<T>&);
  template<class T> valarray<T> log  (const valarray<T>&);
  template<class T> valarray<T> log10(const valarray<T>&);

  template<class T> valarray<T> pow(const valarray<T>&, const valarray<T>&);
  template<class T> valarray<T> pow(const valarray<T>&, const T&);
  template<class T> valarray<T> pow(const T&, const valarray<T>&);

  template<class T> valarray<T> sin  (const valarray<T>&);
  template<class T> valarray<T> sinh (const valarray<T>&);
  template<class T> valarray<T> sqrt (const valarray<T>&);
  template<class T> valarray<T> tan  (const valarray<T>&);
  template<class T> valarray<T> tanh (const valarray<T>&);

  template<typename T> concept_map Range<valarray<T> > see below;
  template<typename T> concept_map Range<const valarray<T> > see below;
}
```

1 The header <valarray> defines five class templates (valarray, slice_array, gslice_array, mask_array, and indirect_array), two classes (slice and gslice), and a series of related function templates for representing and manipulating arrays of values.

2 The valarray array classes are defined to be free of certain forms of aliasing, thus allowing operations on these classes to be optimized.

3 Any function returning a valarray<T> is permitted to return an object of another type, provided all the const member functions of valarray<T> are also applicable to this type. This return type shall not add more than two levels of template nesting over the most deeply nested argument type.[275]

275) Clause 18.2.1 recommends a minimum number of recursively nested template instantiations. This requirement thus indirectly suggests a minimum allowable complexity for valarray expressions.

4 Implementations introducing such replacement types shall provide additional functions and operators as follows:

— for every function taking a `const valarray<T>&`, identical functions taking the replacement types shall be added;

— for every function taking two `const valarray<T>&` arguments, identical functions taking every combination of `const valarray<T>&` and replacement types shall be added.

5 In particular, an implementation shall allow a `valarray<T>` to be constructed from such replacement types and shall allow assignments and computed assignments of such types to `valarray<T>`, `slice_array<T>`, `gslice_array<T>`, `mask_array<T>` and `indirect_array<T>` objects.

6 These library functions are permitted to throw a `bad_alloc` (18.5.2.1) exception if there are not sufficient resources available to carry out the operation. Note that the exception is not mandated.

26.5.2 Class template `valarray` [template.valarray]

```
namespace std {
  template<class T> class valarray {
  public:
    typedef T value_type;

    // 26.5.2.1 construct/destroy:
    valarray();
    explicit valarray(size_t);
    valarray(const T&, size_t);
    valarray(const T*, size_t);
    valarray(const valarray&);
    valarray(valarray&&);
    valarray(const slice_array<T>&);
    valarray(const gslice_array<T>&);
    valarray(const mask_array<T>&);
    valarray(const indirect_array<T>&);
    valarray(initializer_list<T>);
    ~valarray();

    // 26.5.2.2 assignment:
    valarray<T>& operator=(const valarray<T>&);
    valarray<T>& operator=(valarray<T>&&);
    valarray& operator=(initializer_list<T>);
    valarray<T>& operator=(const T&);
    valarray<T>& operator=(const slice_array<T>&);
    valarray<T>& operator=(const gslice_array<T>&);
    valarray<T>& operator=(const mask_array<T>&);
    valarray<T>& operator=(const indirect_array<T>&);

    // 26.5.2.3 element access:
    const T&          operator[](size_t) const;
    T&                operator[](size_t);

    // 26.5.2.4 subset operations:
    valarray<T>       operator[](slice) const;
    slice_array<T>    operator[](slice);
    valarray<T>       operator[](const gslice&) const;
    gslice_array<T>   operator[](const gslice&);
```

```
    valarray<T>       operator[](const valarray<bool>&) const;
    mask_array<T>     operator[](const valarray<bool>&);
    valarray<T>       operator[](const valarray<size_t>&) const;
    indirect_array<T> operator[](const valarray<size_t>&);

    // 26.5.2.5 unary operators:
    valarray<T> operator+() const;
    valarray<T> operator-() const;
    valarray<T> operator~() const;
    valarray<bool> operator!() const;

    // 26.5.2.6 computed assignment:
    valarray<T>& operator*= (const T&);
    valarray<T>& operator/= (const T&);
    valarray<T>& operator%= (const T&);
    valarray<T>& operator+= (const T&);
    valarray<T>& operator-= (const T&);
    valarray<T>& operator^= (const T&);
    valarray<T>& operator&= (const T&);
    valarray<T>& operator|= (const T&);
    valarray<T>& operator<<=(const T&);
    valarray<T>& operator>>=(const T&);

    valarray<T>& operator*= (const valarray<T>&);
    valarray<T>& operator/= (const valarray<T>&);
    valarray<T>& operator%= (const valarray<T>&);
    valarray<T>& operator+= (const valarray<T>&);
    valarray<T>& operator-= (const valarray<T>&);
    valarray<T>& operator^= (const valarray<T>&);
    valarray<T>& operator|= (const valarray<T>&);
    valarray<T>& operator&= (const valarray<T>&);
    valarray<T>& operator<<=(const valarray<T>&);
    valarray<T>& operator>>=(const valarray<T>&);

    // 26.5.2.7 member functions:
    void swap(valarray&&);

    size_t size() const;

    T    sum() const;
    T    min() const;
    T    max() const;

    valarray<T> shift (int) const;
    valarray<T> cshift(int) const;
    valarray<T> apply(T func(T)) const;
    valarray<T> apply(T func(const T&)) const;
    void resize(size_t sz, T c = T());
  };
}
```

1 The class template `valarray<T>` is a one-dimensional smart array, with elements numbered sequentially from zero. It is a representation of the mathematical concept of an ordered set of values. The illusion of higher dimensionality may be produced by the familiar idiom of computed indices, together with the

powerful subsetting capabilities provided by the generalized subscript operators.[276]

2 An implementation is permitted to qualify any of the functions declared in `<valarray>` as `inline`.

26.5.2.1 `valarray` constructors [valarray.cons]

```
valarray();
```

1 *Effects:* Constructs an object of class `valarray<T>`[277] which has zero length.[278]

```
explicit valarray(size_t);
```

2 The array created by this constructor has a length equal to the value of the argument. The elements of the array are constructed using the default constructor for the instantiating type `T`.

```
valarray(const T&, size_t);
```

3 The array created by this constructor has a length equal to the second argument. The elements of the array are initialized with the value of the first argument.

```
valarray(const T*, size_t);
```

4 The array created by this constructor has a length equal to the second argument `n`. The values of the elements of the array are initialized with the first `n` values pointed to by the first argument.[279] If the value of the second argument is greater than the number of values pointed to by the first argument, the behavior is undefined.

```
valarray(const valarray<T>&);
```

5 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array.[280]

```
valarray(valarray<T>&& v);
```

6 The array created by this constructor has the same length as the argument array. The elements are initialized with the values of the corresponding elements of the argument array. After construction, `v` is in a valid but unspecified state.

7 *Complexity:* Constant.

8 *Throws:* Nothing.

```
valarray(initializer_list<T> il);
```

9 *Effects:* Same as `valarray(il.begin(), il.end())`.

```
valarray(const slice_array<T>&);
valarray(const gslice_array<T>&);
valarray(const mask_array<T>&);
valarray(const indirect_array<T>&);
```

276) The intent is to specify an array template that has the minimum functionality necessary to address aliasing ambiguities and the proliferation of temporaries. Thus, the `valarray` template is neither a matrix class nor a field class. However, it is a very useful building block for designing such classes.

277) For convenience, such objects are referred to as "arrays" throughout the remainder of 26.5.

278) This default constructor is essential, since arrays of `valarray` may be useful. The length of an empty array can be increased after initialization with the `resize` member function.

279) This constructor is the preferred method for converting a C array to a `valarray` object.

280) This copy constructor creates a distinct array rather than an alias. Implementations in which arrays share storage are permitted, but they shall implement a copy-on-reference mechanism to ensure that arrays are conceptually distinct.

10 These conversion constructors convert one of the four reference templates to a `valarray`.

```
~valarray();
```

11 The destructor is applied to every element of `*this`; an implementation may return all allocated memory.

26.5.2.2 `valarray` assignment [valarray.assign]

```
valarray<T>& operator=(const valarray<T>&);
```

1 Each element of the `*this` array is assigned the value of the corresponding element of the argument array. The resulting behavior is undefined if the length of the argument array is not equal to the length of the `*this` array.

```
valarray<T>& operator=(valarray<T>&& v);
```

2 *Effects:* `*this` obtains the value of `v`. After the assignment, `v` is in a valid but unspecified state.

3 *Complexity:* Constant.

4 *Throws:* Nothing.

```
valarray& operator=(initializer_list<T> il);
```

5 *Effects:* `*this = valarray(il)`.

6 *Returns:* `*this`.

```
valarray<T>& operator=(const T&);
```

7 The scalar assignment operator causes each element of the `*this` array to be assigned the value of the argument.

```
valarray<T>& operator=(const slice_array<T>&);
valarray<T>& operator=(const gslice_array<T>&);
valarray<T>& operator=(const mask_array<T>&);
valarray<T>& operator=(const indirect_array<T>&);
```

8 *Requires:* The length of the array to which the argument refers equals `size()`.

9 These operators allow the results of a generalized subscripting operation to be assigned directly to a `valarray`.

10 If the value of an element in the left-hand side of a valarray assignment operator depends on the value of another element in that left-hand side, the resulting behavior is undefined.

26.5.2.3 `valarray` element access [valarray.access]

```
const T&  operator[](size_t) const;
T& operator[](size_t);
```

1 The subscript operator returns a reference to the corresponding element of the array.

2 Thus, the expression `(a[i] = q, a[i]) == q` evaluates as true for any non-constant `valarray<T> a`, any `T q`, and for any `size_t i` such that the value of `i` is less than the length of `a`.

3 The expression `&a[i+j] == &a[i] + j` evaluates as true for all `size_t i` and `size_t j` such that `i+j` is less than the length of the array `a`.

4 Likewise, the expression `&a[i] != &b[j]` evaluates as `true` for any two arrays `a` and `b` and for any `size_t i` and `size_t j` such that `i` is less than the length of `a` and `j` is less than the length of `b`. This property indicates an absence of aliasing and may be used to advantage by optimizing compilers.[281]

5 The reference returned by the subscript operator for an array is guaranteed to be valid until the member function `resize(size_t, T)` (26.5.2.7) is called for that array or until the lifetime of that array ends, whichever happens first.

6 If the subscript operator is invoked with a `size_t` argument whose value is not less than the length of the array, the behavior is undefined.

26.5.2.4 `valarray` subset operations [valarray.sub]

```
valarray<T>       operator[](slice) const;
slice_array<T>    operator[](slice);
valarray<T>       operator[](const gslice&) const;
gslice_array<T>   operator[](const gslice&);
valarray<T>       operator[](const valarray<bool>&) const;
mask_array<T>     operator[](const valarray<bool>&);
valarray<T>       operator[](const valarray<size_t>&) const;
indirect_array<T> operator[](const valarray<size_t>&);
```

1 Each of these operations returns a subset of the array. The `const`-qualified versions return this subset as a new `valarray`. The non-`const` versions return a class template object which has reference semantics to the original array.

26.5.2.5 `valarray` unary operators [valarray.unary]

```
valarray<T> operator+() const;
valarray<T> operator-() const;
valarray<T> operator~() const;
valarray<bool> operator!() const;
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` (`bool` for `operator!`) or which may be unambiguously implicitly converted to type `T` (`bool` for `operator!`).

2 Each of these operators returns an array whose length is equal to the length of the array. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array.

26.5.2.6 `valarray` computed assignment [valarray.cassign]

```
valarray<T>& operator*= (const valarray<T>&);
valarray<T>& operator/= (const valarray<T>&);
valarray<T>& operator%= (const valarray<T>&);
valarray<T>& operator+= (const valarray<T>&);
valarray<T>& operator-= (const valarray<T>&);
valarray<T>& operator^= (const valarray<T>&);
valarray<T>& operator&= (const valarray<T>&);
valarray<T>& operator|= (const valarray<T>&);
valarray<T>& operator<<=(const valarray<T>&);
```

281) Compilers may take advantage of inlining, constant propagation, loop fusion, tracking of pointers obtained from `operator new`, and other techniques to generate efficient `valarrays`.

```
valarray<T>& operator>>=(const valarray<T>&);
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied. Each of these operators performs the indicated operation on each of its elements and the corresponding element of the argument array.

2 The array is then returned by reference.

3 If the array and the argument array do not have the same length, the behavior is undefined. The appearance of an array on the left-hand side of a computed assignment does `not` invalidate references or pointers.

4 If the value of an element in the left-hand side of a valarray computed assignment operator depends on the value of another element in that left hand side, the resulting behavior is undefined.

```
valarray<T>& operator*= (const T&);
valarray<T>& operator/= (const T&);
valarray<T>& operator%= (const T&);
valarray<T>& operator+= (const T&);
valarray<T>& operator-= (const T&);
valarray<T>& operator^= (const T&);
valarray<T>& operator&= (const T&);
valarray<T>& operator|= (const T&);
valarray<T>& operator<<=(const T&);
valarray<T>& operator>>=(const T&);
```

5 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied.

6 Each of these operators applies the indicated operation to each element of the array and the non-array argument.

7 The array is then returned by reference.

8 The appearance of an array on the left-hand side of a computed assignment does *not* invalidate references or pointers to the elements of the array.

26.5.2.7 `valarray` member functions [valarray.members]

```
void swap(valarray&& v);
```

1 *Effects:* `*this` obtains the value of `v`. `v` obtains the value of `*this`.

2 *Complexity:* Constant.

3 *Throws:* Nothing.

```
size_t size() const;
```

4 This function returns the number of elements in the array.

```
T sum() const;
```

This function may only be instantiated for a type `T` to which `operator+=` can be applied. This function returns the sum of all the elements of the array.

5 If the array has length 0, the behavior is undefined. If the array has length 1, `sum()` returns the value of element 0. Otherwise, the returned value is calculated by applying `operator+=` to a copy of an element of the array and all other elements of the array in an unspecified order.

```
T min() const;
```

6 This function returns the minimum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

```
T max() const;
```

7 This function returns the maximum value contained in `*this`. The value returned for an array of length 0 is undefined. For an array of length 1, the value of element 0 is returned. For all other array lengths, the determination is made using `operator<`.

```
valarray<T> shift(int n) const;
```

8 This function returns an object of class `valarray<T>` of length `size()`, each of whose elements I is `(*this)[I + n]` if `I + n` is non-negative and less than `size()`, otherwise `T()`. Thus if element zero is taken as the leftmost element, a positive value of `n` shifts the elements left `n` places, with zero fill.

9 [*Example:* If the argument has the value -2, the first two elements of the result will be constructed using the default constructor; the third element of the result will be assigned the value of the first element of the argument; etc. — *end example*]

```
valarray<T> cshift(int n) const;
```

10 This function returns an object of class `valarray<T>` of length `size()` that is a circular shift of `*this`. If element zero is taken as the leftmost element, a non-negative value of n shifts the elements circularly left n places and a negative value of n shifts the elements circularly right -n places.

```
valarray<T> apply(T func(T)) const;
valarray<T> apply(T func(const T&)) const;
```

11 These functions return an array whose length is equal to the array. Each element of the returned array is assigned the value returned by applying the argument function to the corresponding element of the array.

```
void resize(size_t sz, T c = T());
```

12 This member function changes the length of the `*this` array to `sz` and then assigns to each element the value of the second argument. Resizing invalidates all pointers and references to elements in the array.

26.5.3 `valarray` non-member operations [valarray.nonmembers]

26.5.3.1 `valarray` binary operators [valarray.binary]

```
template<class T> valarray<T> operator*
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator/
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator%
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator+
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator-
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator^
    (const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<T> operator&
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator|
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator<<
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> operator>>
    (const valarray<T>&, const valarray<T>&);
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

2 Each of these operators returns an array whose length is equal to the lengths of the argument arrays. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the argument arrays do not have the same length, the behavior is undefined.

```
template<class T> valarray<T> operator* (const valarray<T>&, const T&);
template<class T> valarray<T> operator* (const T&, const valarray<T>&);
template<class T> valarray<T> operator/ (const valarray<T>&, const T&);
template<class T> valarray<T> operator/ (const T&, const valarray<T>&);
template<class T> valarray<T> operator% (const valarray<T>&, const T&);
template<class T> valarray<T> operator% (const T&, const valarray<T>&);
template<class T> valarray<T> operator+ (const valarray<T>&, const T&);
template<class T> valarray<T> operator+ (const T&, const valarray<T>&);
template<class T> valarray<T> operator- (const valarray<T>&, const T&);
template<class T> valarray<T> operator- (const T&, const valarray<T>&);
template<class T> valarray<T> operator^ (const valarray<T>&, const T&);
template<class T> valarray<T> operator^ (const T&, const valarray<T>&);
template<class T> valarray<T> operator& (const valarray<T>&, const T&);
template<class T> valarray<T> operator& (const T&, const valarray<T>&);
template<class T> valarray<T> operator| (const valarray<T>&, const T&);
template<class T> valarray<T> operator| (const T&, const valarray<T>&);
template<class T> valarray<T> operator<<(const valarray<T>&, const T&);
template<class T> valarray<T> operator<<(const T&, const valarray<T>&);
template<class T> valarray<T> operator>>(const valarray<T>&, const T&);
template<class T> valarray<T> operator>>(const T&, const valarray<T>&);
```

4 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

5 Each of these operators returns an array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array argument and the non-array argument.

26.5.3.2 `valarray` logical operators [valarray.comparison]

```
template<class T> valarray<bool> operator==
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator!=
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<
    (const valarray<T>&, const valarray<T>&);
```

```
template<class T> valarray<bool> operator>
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator<=
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator>=
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator&&
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<bool> operator||
    (const valarray<T>&, const valarray<T>&);
```

1 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `bool` or which can be unambiguously implicitly converted to type `bool`.

2 Each of these operators returns a `bool` array whose length is equal to the length of the array arguments. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding elements of the argument arrays.

3 If the two array arguments do not have the same length, the behavior is undefined.

```
template<class T> valarray<bool> operator==(const valarray<T>&, const T&);
template<class T> valarray<bool> operator==(const T&, const valarray<T>&);
template<class T> valarray<bool> operator!=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator!=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator< (const valarray<T>&, const T&);
template<class T> valarray<bool> operator< (const T&, const valarray<T>&);
template<class T> valarray<bool> operator> (const valarray<T>&, const T&);
template<class T> valarray<bool> operator> (const T&, const valarray<T>&);
template<class T> valarray<bool> operator<=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator<=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator>=(const valarray<T>&, const T&);
template<class T> valarray<bool> operator>=(const T&, const valarray<T>&);
template<class T> valarray<bool> operator&&(const valarray<T>&, const T&);
template<class T> valarray<bool> operator&&(const T&, const valarray<T>&);
template<class T> valarray<bool> operator||(const valarray<T>&, const T&);
template<class T> valarray<bool> operator||(const T&, const valarray<T>&);
```

4 Each of these operators may only be instantiated for a type `T` to which the indicated operator can be applied and for which the indicated operator returns a value which is of type `bool` or which can be unambiguously implicitly converted to type `bool`.

5 Each of these operators returns a `bool` array whose length is equal to the length of the array argument. Each element of the returned array is initialized with the result of applying the indicated operator to the corresponding element of the array and the non-array argument.

26.5.3.3 `valarray` transcendentals [valarray.transcend]

```
template<class T> valarray<T> abs  (const valarray<T>&);
template<class T> valarray<T> acos (const valarray<T>&);
template<class T> valarray<T> asin (const valarray<T>&);
template<class T> valarray<T> atan (const valarray<T>&);
template<class T> valarray<T> atan2
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> atan2(const valarray<T>&, const T&);
template<class T> valarray<T> atan2(const T&, const valarray<T>&);
```

```
template<class T> valarray<T> cos  (const valarray<T>&);
template<class T> valarray<T> cosh (const valarray<T>&);
template<class T> valarray<T> exp  (const valarray<T>&);
template<class T> valarray<T> log  (const valarray<T>&);
template<class T> valarray<T> log10(const valarray<T>&);
template<class T> valarray<T> pow
    (const valarray<T>&, const valarray<T>&);
template<class T> valarray<T> pow  (const valarray<T>&, const T&);
template<class T> valarray<T> pow  (const T&, const valarray<T>&);
template<class T> valarray<T> sin  (const valarray<T>&);
template<class T> valarray<T> sinh (const valarray<T>&);
template<class T> valarray<T> sqrt (const valarray<T>&);
template<class T> valarray<T> tan  (const valarray<T>&);
template<class T> valarray<T> tanh (const valarray<T>&);
```

1 Each of these functions may only be instantiated for a type `T` to which a unique function with the indicated name can be applied (unqualified). This function shall return a value which is of type `T` or which can be unambiguously implicitly converted to type `T`.

26.5.3.4 `valarray` specialized algorithms [valarray.special]

```
template <class T> void swap(valarray<T>& x, valarray<T>& y);
template <class T> void swap(valarray<T>&& x, valarray<T>& y);
template <class T> void swap(valarray<T>& x, valarray<T>&& y);
```

1 *Effects:* `x.swap(y)`.

26.5.4 Class `slice` [class.slice]

```
namespace std {
  class slice {
  public:
    slice();
    slice(size_t, size_t, size_t);

    size_t start() const;
    size_t size() const;
    size_t stride() const;
  };
}
```

1 The `slice` class represents a BLAS-like slice from an array. Such a slice is specified by a starting index, a length, and a stride.[282]

26.5.4.1 `slice` constructors [cons.slice]

```
slice();
slice(size_t start, size_t length, size_t stride);
slice(const slice&);
```

282) BLAS stands for *Basic Linear Algebra Subprograms.* C++ programs may instantiate this class. See, for example, Dongarra, Du Croz, Duff, and Hammerling: *A set of Level 3 Basic Linear Algebra Subprograms*; Technical Report MCS-P1-0888, Argonne National Laboratory (USA), Mathematics and Computer Science Division, August, 1988.

1 The default constructor is equivalent to `slice(0, 0, 0)`. A default constructor is provided only to permit the declaration of arrays of slices. The constructor with arguments for a slice takes a start, length, and stride parameter.

2 [*Example:* `slice(3, 8, 2)` constructs a slice which selects elements 3, 5, 7, ... 17 from an array. — *end example*]

26.5.4.2 `slice` access functions [slice.access]

```
size_t start() const;
size_t size() const;
size_t stride() const;
```

1 These functions return the start, length, or stride specified by a `slice` object.

26.5.5 Class template `slice_array` [template.slice.array]

```
namespace std {
  template <class T> class slice_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    slice_array(const slice_array&);
   ~slice_array();
    const slice_array& operator=(const slice_array&) const;
  void operator=(const T&) const;

    slice_array() = delete;       // as implied by declaring copy constructor above
  };
}
```

1 The `slice_array` template is a helper template used by the `slice` subscript operator

```
slice_array<T> valarray<T>::operator[](slice);
```

It has reference semantics to a subset of an array specified by a `slice` object.

2 [*Example:* The expression `a[slice(1, 5, 3)] = b;` has the effect of assigning the elements of `b` to a slice of the elements in `a`. For the slice shown, the elements selected from `a` are 1, 4, ..., 13. — *end example*]

26.5.5.1 `slice_array` assignment [slice.arr.assign]

```
void operator=(const valarray<T>&) const;
const slice_array& operator=(const slice_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.5.5.2 `slice_array` computed assignment [slice.arr.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `slice_array` object refers.

26.5.5.3 `slice_array` fill function [slice.arr.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `slice_array` object refers.

26.5.6 The `gslice` class [class.gslice]

```
namespace std {
  class gslice {
  public:
    gslice();
    gslice(size_t s, const valarray<size_t>& l, const valarray<size_t>& d);

    size_t           start() const;
    valarray<size_t> size() const;
    valarray<size_t> stride() const;
  };
}
```

1 This class represents a generalized slice out of an array. A `gslice` is defined by a starting offset (s), a set of lengths (l_j), and a set of strides (d_j). The number of lengths shall equal the number of strides.

2 A `gslice` represents a mapping from a set of indices (i_j), equal in number to the number of strides, to a single index k. It is useful for building multidimensional array classes using the `valarray` template, which is one-dimensional. The set of one-dimensional index values specified by a `gslice` are

$$k = s + \sum_j i_j d_j$$

where the multidimensional indices i_j range in value from 0 to $l_{ij} - 1$.

3 [*Example:* The `gslice` specification

```
start  = 3
length = {2, 4, 3}
stride = {19, 4, 1}
```

yields the sequence of one-dimensional indices

$$k = 3 + (0,1) \times 19 + (0,1,2,3) \times 4 + (0,1,2) \times 1$$

which are ordered as shown in the following table:

$$\begin{array}{cccc} (i_0, & i_1, & i_2, & k) = \\ & (0, & 0, & 0, & 3), \\ & (0, & 0, & 1, & 4), \\ & (0, & 0, & 2, & 5), \\ & (0, & 1, & 0, & 7), \\ & (0, & 1, & 1, & 8), \\ & (0, & 1, & 2, & 9), \\ & (0, & 2, & 0, & 11), \\ & (0, & 2, & 1, & 12), \\ & (0, & 2, & 2, & 13), \\ & (0, & 3, & 0, & 15), \\ & (0, & 3, & 1, & 16), \\ & (0, & 3, & 2, & 17), \\ & (1, & 0, & 0, & 22), \\ & (1, & 0, & 1, & 23), \\ & \ldots & & & \\ & (1, & 3, & 2, & 36) \end{array}$$

That is, the highest-ordered index turns fastest. — *end example*]

4 It is possible to have degenerate generalized slices in which an address is repeated.

5 [*Example:* If the stride parameters in the previous example are changed to {1, 1, 1}, the first few elements of the resulting sequence of indices will be

$$\begin{array}{cccc} & (0, & 0, & 0, & 3), \\ & (0, & 0, & 1, & 4), \\ & (0, & 0, & 2, & 5), \\ & (0, & 1, & 0, & 4), \\ & (0, & 1, & 1, & 5), \\ & (0, & 1, & 2, & 6), \\ & \ldots & & & \end{array}$$

— *end example*]

6 If a degenerate slice is used as the argument to the non-`const` version of `operator[](const gslice&)`, the resulting behavior is undefined.

26.5.6.1 `gslice` constructors [gslice.cons]

```
gslice();
gslice(size_t start, const valarray<size_t>& lengths,
         const valarray<size_t>& strides);
gslice(const gslice&);
```

1 The default constructor is equivalent to `gslice(0, valarray<size_t>(), valarray<size_t>())`. The constructor with arguments builds a `gslice` based on a specification of start, lengths, and strides, as explained in the previous section.

26.5.6.2 `gslice` access functions [gslice.access]

```
size_t           start()  const;
valarray<size_t> size() const;
valarray<size_t> stride() const;
```

1 These access functions return the representation of the start, lengths, or strides specified for the `gslice`.

26.5.7 Class template `gslice_array` [template.gslice.array]

```
namespace std {
  template <class T> class gslice_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    gslice_array(const gslice_array&);
    ~gslice_array();
    const gslice_array& operator=(const gslice_array&) const;
    void operator=(const T&) const;

    gslice_array() = delete;       // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the `slice` subscript operator

```
gslice_array<T> valarray<T>::operator[](const gslice&);
```

2 It has reference semantics to a subset of an array specified by a `gslice` object.

3 Thus, the expression `a[gslice(1, length, stride)] = b` has the effect of assigning the elements of `b` to a generalized slice of the elements in `a`.

26.5.7.1 `gslice_array` assignment [gslice.array.assign]

```
void operator=(const valarray<T>&) const;
const gslice_array& operator=(const gslice_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which the `gslice_array` refers.

26.5.7.2 `gslice_array` [gslice.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.5.7.3 `gslice_array` fill function [gslice.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `gslice_array` object refers.

26.5.8 Class template `mask_array` [template.mask.array]

```
namespace std {
  template <class T> class mask_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    mask_array(const mask_array&);
   ~mask_array();
    const mask_array& operator=(const mask_array&) const;
    void operator=(const T&) const;

    mask_array() = delete;          // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the mask subscript operator:

`mask_array<T> valarray<T>::operator[](const valarray<bool>&).`

2 It has reference semantics to a subset of an array specified by a boolean mask. Thus, the expression `a[mask] = b;` has the effect of assigning the elements of `b` to the masked elements in `a` (those for which the corresponding element in `mask` is `true`.)

26.5.8.1 `mask_array` assignment [mask.array.assign]

```
void operator=(const valarray<T>&) const;
const mask_array& operator=(const mask_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

26.5.8.2 `mask_array` computed assignment [mask.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the mask object refers.

26.5.8.3 `mask_array` fill function [mask.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `mask_array` object refers.

26.5.9 Class template `indirect_array` [template.indirect.array]

```
namespace std {
  template <class T> class indirect_array {
  public:
    typedef T value_type;

    void operator=  (const valarray<T>&) const;
    void operator*= (const valarray<T>&) const;
    void operator/= (const valarray<T>&) const;
    void operator%= (const valarray<T>&) const;
    void operator+= (const valarray<T>&) const;
    void operator-= (const valarray<T>&) const;
    void operator^= (const valarray<T>&) const;
    void operator&= (const valarray<T>&) const;
```

```
    void operator|= (const valarray<T>&) const;
    void operator<<=(const valarray<T>&) const;
    void operator>>=(const valarray<T>&) const;

    indirect_array(const indirect_array&);
   ~indirect_array();
    const indirect_array& operator=(const indirect_array&) const;
    void operator=(const T&) const;

    indirect_array() = delete;          // as implied by declaring copy constructor above
  };
}
```

1 This template is a helper template used by the indirect subscript operator

`indirect_array<T> valarray<T>::operator[](const valarray<size_t>&).`

2 It has reference semantics to a subset of an array specified by an `indirect_array`. Thus the expression `a[indirect] = b;` has the effect of assigning the elements of `b` to the elements in `a` whose indices appear in `indirect`.

26.5.9.1 `indirect_array` assignment [indirect.array.assign]

```
void operator=(const valarray<T>&) const;
const indirect_array& operator=(const indirect_array&) const;
```

1 These assignment operators have reference semantics, assigning the values of the argument array elements to selected elements of the `valarray<T>` object to which it refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

3 [*Example:*

```
int addr[] = {2, 3, 1, 4, 4};
valarray<size_t> indirect(addr, 5);
valarray<double> a(0., 10), b(1., 5);
a[indirect] = b;
```

results in undefined behavior since element 4 is specified twice in the indirection. — *end example*]

26.5.9.2 `indirect_array` computed assignment [indirect.array.comp.assign]

```
void operator*= (const valarray<T>&) const;
void operator/= (const valarray<T>&) const;
void operator%= (const valarray<T>&) const;
void operator+= (const valarray<T>&) const;
void operator-= (const valarray<T>&) const;
void operator^= (const valarray<T>&) const;
void operator&= (const valarray<T>&) const;
void operator|= (const valarray<T>&) const;
void operator<<=(const valarray<T>&) const;
void operator>>=(const valarray<T>&) const;
```

1 These computed assignments have reference semantics, applying the indicated operation to the elements of the argument array and selected elements of the `valarray<T>` object to which the `indirect_array` object refers.

2 If the `indirect_array` specifies an element in the `valarray<T>` object to which it refers more than once, the behavior is undefined.

26.5.9.3 `indirect_array` fill function [indirect.array.fill]

```
void operator=(const T&) const;
```

1 This function has reference semantics, assigning the value of its argument to the elements of the `valarray<T>` object to which the `indirect_array` object refers.

26.5.9.4 Valarray concept maps [valarray.concepts]

```
template<typename T>
concept_map Range<valarray<T> > {
  typedef unspecified  iterator;
  iterator begin(valarray<T>& a);
  iterator end(valarray<T>& a);
}

template<typename T>
concept_map Range<const valarray<T> > {
  typedef unspecified  iterator;
  iterator begin(const valarray<T>& a);
  iterator end(const valarray<T>& a);
}
```

1 *Note:* these concept_maps adapt `valarray` to the `Range` concept.

```
typedef unspecified  iterator;
```

2 *Requires:* `iterator` shall meet the requirements of the `RandomAccessIterator` concept and `iterator::reference` shall equal `const T&`.

```
iterator begin(valarray<T>& a);
iterator begin(const valarray<T>& a);
```

3 *Returns:* an iterator referencing the first value in the numeric array.

```
iterator end(valarray<T>& a);
iterator end(const valarray<T>& a);
```

4 *Returns:* an iterator referencing one past the last value in the numeric array.

26.6 Generalized numeric operations [numeric.ops]

Header `<numeric>` synopsis

```
namespace std {
  template <InputIterator Iter, MoveConstructible T>
    requires HasPlus<T, Iter::reference>
          && HasAssign<T, HasPlus<T, Iter::reference>::result_type>
    T accumulate(Iter first, Iter last, T init);
  template <InputIterator Iter, MoveConstructible T,
```

```
          Callable<auto, const T&, Iter::reference> BinaryOperation>
  requires HasAssign<T, BinaryOperation::result_type>
        && CopyConstructible<BinaryOperation>
  T accumulate(Iter first, Iter last, T init,
               BinaryOperation binary_op);
template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T>
  requires HasMultiply<Iter1::reference, Iter2::reference>
        && HasPlus<T, HasMultiply<Iter1::reference, Iter2::reference>::result_type>
        && HasAssign<
             T,
             HasPlus<T,
                     HasMultiply<Iter1::reference, Iter2::reference>::result_type>::result_type>
  T inner_product(Iter1 first1, Iter1 last1,
                  Iter2 first2, T init);
template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T,
          class BinaryOperation1,
          Callable<auto, Iter1::reference, Iter2::reference> BinaryOperation2>
  requires Callable<BinaryOperation1, const T&, BinaryOperation2::result_type>
        && HasAssign<T, BinaryOperation1::result_type>
        && CopyConstructible<BinaryOperation1>
        && CopyConstructible<BinaryOperation2>
  T inner_product(Iter1 first1, Iter1 last1,
                  Iter2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
  requires HasPlus<InIter::value_type, InIter::reference>
        && HasAssign<InIter::value_type,
                     HasPlus<InIter::value_type, InIter::reference>::result_type>
        && Constructible<InIter::value_type, InIter::reference>
  OutIter partial_sum(InIter first, InIter last,
                      OutIter result);
template<InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
         Callable<auto, const InIter::value_type&, InIter::reference> BinaryOperation>
  requires HasAssign<InIter::value_type, BinaryOperation::result_type>
        && Constructible<InIter::value_type, InIter::reference>
        && CopyConstructible<BinaryOperation>
  OutIter partial_sum(InIter first, InIter last,
                      OutIter result, BinaryOperation binary_op);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
  requires HasMinus<InIter::value_type, InIter::value_type>
        && Constructible<InIter::value_type, InIter::reference>
        && OutputIterator<OutIter, HasMinus<InIter::value_type, InIter::value_type>::result_type>
        && MoveAssignable<InIter::value_type>
  OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result);
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
          Callable<auto, const InIter::value_type&, const InIter::value_type&> BinaryOperation>
  requires Constructible<InIter::value_type, InIter::reference>
        && OutputIterator<OutIter, BinaryOperation::result_type>
        && MoveAssignable<InIter::value_type>
        && CopyConstructible<BinaryOperation>
  OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result,
                              BinaryOperation binary_op);
```

```
  template <ForwardIterator Iter, HasPreincrement T>
    requires OutputIterator<Iter, const T&>
    void iota(Iter first, Iter last, T value);
```

1 The requirements on the types of algorithms' arguments that are described in the introduction to clause 25 also apply to the following algorithms.

26.6.1 Accumulate [accumulate]

```
template <InputIterator Iter, MoveConstructible T>
  requires HasPlus<T, Iter::reference>
        && HasAssign<T, HasPlus<T, Iter::reference>::result_type>
  T accumulate(Iter first, Iter last, T init);
template <InputIterator Iter, MoveConstructible T,
          Callable<auto, const T&, Iter::reference> BinaryOperation>
  requires HasAssign<T, BinaryOperation::result_type>
        && CopyConstructible<BinaryOperation>
  T accumulate(Iter first, Iter last, T init,
               BinaryOperation binary_op);
```

1 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifies it with `acc = acc + *i` or `acc = binary_op(acc, *i)` for every iterator `i` in the range `[first,last)` in order.[283]

2 *Requires:* In the range `[first,last]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[284]

26.6.2 Inner product [inner.product]

```
template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T>
  requires HasMultiply<Iter1::reference, Iter2::reference>
        && HasPlus<T, HasMultiply<Iter1::reference, Iter2::reference>::result_type>
        && HasAssign<
             T,
             HasPlus<T,
                     HasMultiply<Iter1::reference, Iter2::reference>::result_type>::result_type>
  T inner_product(Iter1 first1, Iter1 last1,
                  Iter2 first2, T init);
template <InputIterator Iter1, InputIterator Iter2, MoveConstructible T,
          class BinaryOperation1,
          Callable<auto, Iter1::reference, Iter2::reference> BinaryOperation2>
  requires Callable<BinaryOperation1, const T&, BinaryOperation2::result_type>
        && HasAssign<T, BinaryOperation1::result_type>
        && CopyConstructible<BinaryOperation1>
        && CopyConstructible<BinaryOperation2>
  T inner_product(Iter1 first1, Iter1 last1,
                  Iter2 first2, T init,
                  BinaryOperation1 binary_op1,
                  BinaryOperation2 binary_op2);
```

283) `accumulate` is similar to the APL reduction operator and Common Lisp reduce function, but it avoids the difficulty of defining the result of reduction on an empty sequence by always requiring an initial value.
284) The use of fully closed ranges is intentional

1 *Effects:* Computes its result by initializing the accumulator `acc` with the initial value `init` and then modifying it with `acc = acc + (*i1) * (*i2)` or `acc = binary_op1(acc, binary_op2(*i1, *i2))` for every iterator `i1` in the range `[first,last)` and iterator `i2` in the range `[first2,first2 + (last - first))` in order.

2 *Requires:* In the ranges `[first,last]` and `[first2,first2 + (last - first)]` `binary_op1` and `binary_op2` shall neither modify elements nor invalidate iterators or subranges.[285]

26.6.3 Partial sum [partial.sum]

```
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
  requires HasPlus<InIter::value_type, InIter::reference>
        && HasAssign<InIter::value_type,
                     HasPlus<InIter::value_type, InIter::reference>::result_type>
        && Constructible<InIter::value_type, InIter::reference>
  OutIter partial_sum(InIter first, InIter last,
                      OutIter result);
template<InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
         Callable<auto, const InIter::value_type&, InIter::reference> BinaryOperation>
  requires HasAssign<InIter::value_type, BinaryOperation::result_type>
        && Constructible<InIter::value_type, InIter::reference>
        && CopyConstructible<BinaryOperation>
  OutIter partial_sum(InIter first, InIter last,
                      OutIter result, BinaryOperation binary_op);
```

1 *Effects:* Assigns to every element referred to by iterator `i` in the range `[result,result + (last - first))` a value correspondingly equal to

```
((...(*first + *(first + 1)) + ...) + *(first + (i - result)))
```

or

```
binary_op(binary_op(...,
    binary_op(*first, *(first + 1)),...), *(first + (i - result)))
```

2 *Returns:* `result + (last - first)`.

3 *Complexity:* Exactly `(last - first) - 1` applications of `binary_op`.

4 *Requires:* In the ranges `[first,last]` and `[result,result + (last - first)]` `binary_op` shall neither modify elements nor invalidate iterators or subranges.[286]

5 *Remarks:* `result` may be equal to `first`.

26.6.4 Adjacent difference [adjacent.difference]

```
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter>
  requires HasMinus<InIter::value_type, InIter::value_type>
        && Constructible<InIter::value_type, InIter::reference>
        && OutputIterator<OutIter, HasMinus<InIter::value_type, InIter::value_type>::result_type>
        && MoveAssignable<InIter::value_type>
  OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result);
```

285) The use of fully closed ranges is intentional

286) The use of fully closed ranges is intentional.

```
template <InputIterator InIter, OutputIterator<auto, const InIter::value_type&> OutIter,
          Callable<auto, const InIter::value_type&, const InIter::value_type&> BinaryOperation>
  requires Constructible<InIter::value_type, InIter::reference>
        && OutputIterator<OutIter, BinaryOperation::result_type>
        && MoveAssignable<InIter::value_type>
        && CopyConstructible<BinaryOperation>
  OutIter adjacent_difference(InIter first, InIter last,
                              OutIter result,
                              BinaryOperation binary_op);
```

1 *Effects:* Assigns to every element referred to by iterator `i` in the range `[result + 1,result + (last - first))` a value correspondingly equal to

```
*(first + (i - result)) - *(first + (i - result) - 1)
```

or

```
binary_op(*(first + (i - result)), *(first + (i - result) - 1)).
```

`result` gets the value of `*first`.

2 *Requires:* In the ranges `[first,last]` and `[result,result + (last - first)]`, `binary_op` shall neither modify elements nor invalidate iterators or subranges.[287]

3 *Remarks:* `result` may be equal to `first`.

4 *Returns:* `result + (last - first)`.

5 *Complexity:* Exactly `(last - first) - 1` applications of `binary_op`.

26.6.5 Iota [numeric.iota]

```
template <ForwardIterator Iter, HasPreincrement T>
  requires OutputIterator<Iter, const T&>
  void iota(Iter first, Iter last, T value);
```

1 *Effects:* For each element referred to by the iterator `i` in the range `[first,last)`, assigns `*i = value` and increments value as if by `++value`.

2 *Complexity:* Exactly `last - first` increments and assignments.

26.7 C Library [c.math]

1 The header `<ctgmath>` simply includes the headers `<ccomplex>` and `<cmath>`.

2 [*Note:* The overloads provided in C99 by magic macros are already provided in `<ccomplex>` and `<cmath>` by "sufficient" additional overloads. — *end note*]

3 The header `<tgmath.h>` effectively includes the headers `<complex.h>` and `<math.h>`.

4 Tables 97 and 98 describe headers `<cmath>`[288] and `<cstdlib>`, respectively.

5 The contents of these headers are the same as the Standard C library headers `<math.h>` and `<stdlib.h>` respectively, with the following changes:

287) The use of fully closed ranges is intentional.
288) all macros except `HUGE_VAL`, both types, many functions, and all templates added by TR1.

Table 97 — Header <cmath> synopsis

Type	Name(s)			
Macros:				
FP_FAST_FMA	FP_ILOGBNAN	FP_SUBNORMAL	HUGE_VALL	MATH_ERRNO
FP_FAST_FMAF	FP_INFINITE	FP_ZERO	INFINITY	MATH_ERREXCEPT
FP_FAST_FMAL	FP_NAN	HUGE_VAL	NAN	math_errhandling
FP_ILOGB0	FP_NORMAL	HUGE_VALF		
Types:	double_t	float_t		
Functions:				
abs	cosh	fmod	logb	remquo
acos	erf	frexp	lrint	rint
acosh	erfc	hypot	lround	round
asin	exp2	ilogb	modf	scalbln
asinh	exp	ldexp	nan	scalbn
atan	expm1	lgamma	nanf	sin
atan2	fabs	llrint	nanl	sinh
atanh	fdim	llround	nearbyint	sqrt
cbrt	floor	log	nextafter	tan
ceil	fma	log10	nexttoward	tanh
copysign	fmax	log1p	pow	tgamma
cos	fmin	log2	remainder	trunc
Templates:				
fpclassify	isgreaterequal	islessequal	isnan	isunordered
isfinite	isinf	islessgreater	isnormal	signbit
isgreater	isless			

Table 98 — Header <cstdlib> synopsis

Type	Name(s)	
Macro:	RAND_MAX	
Types:		
div_t	ldiv_t	lldiv_t
Functions:		
abs	ldiv	rand
div	llabs	srand
labs	lldiv	

6 The `rand` function has the semantics specified in the C standard, except that the implementation may specify that particular library functions may call `rand`. It is implementation defined whether the `rand` function may introduce data races (17.6.5.7). [*Note:* The random number generation (26.4) facilities in this standard are often preferable to `rand`. — *end note*]

7 In addition to the `int` versions of certain math functions in `<cstdlib>`, C++ adds `long` and `long long` overloaded versions of these functions, with the same semantics.

8 The added signatures are:

```
long abs(long);                     // labs()
long long abs(long long);           // llabs()
ldiv_t div(long, long);             // ldiv()
lldiv_t div(long long, long long);  // lldiv()
```

9 In addition to the `double` versions of the math functions in `<cmath>`, C++ adds `float` and `long double` overloaded versions of these functions, with the same semantics.

10 The added signatures are:

```
float abs(float);
float acos(float);
float acosh(float);
float asin(float);
float asinh(float);
float atan(float);
float atan2(float, float);
float atanh(float);
float cbrt(float);
float ceil(float);
float copysign(float, float);
float cos(float);
float cosh(float);
float erf(float);
float erfc(float);
float exp(float);
float exp2(float);
float expm1(float);
float fabs(float);
float fdim(float, float);
float floor(float);
float fma(float, float, float);
float fmax(float, float);
float fmin(float, float);
float fmod(float, float);
float frexp(float, int*);
float hypot(float, float);
int ilogb(float);
float ldexp(float, int);
float lgamma(float);
long long llrint(float);
long long llround(float);
float log(float);
float log10(float);
float log1p(float);
float log2(float);
float logb(float);
```

```
long lrint(float);
long lround(float);
float modf(float, float*);
float nearbyint(float);
float nextafter(float, float);
float nexttoward(float, long double);
float pow(float, float);
float remainder(float, float);
float remquo(float, float, int *);
float rint(float);
float round(float);
float scalbln(float, long);
float scalbn(float, int);
float sin(float);
float sinh(float);
float sqrt(float);
float tan(float);
float tanh(float);
float tgamma(float);
float trunc(float);

double abs(double);            // fabs()

long double abs(long double);
long double acos(long double);
long double acosh(long double);
long double asin(long double);
long double asinh(long double);
long double atan(long double);
long double atan2(long double, long double);
long double atanh(long double);
long double cbrt(long double);
long double ceil(long double);
long double copysign(long double, long double);
long double cos(long double);
long double cosh(long double);
long double erf(long double);
long double erfc(long double);
long double exp(long double);
long double exp2(long double);
long double expm1(long double);
long double fabs(long double);
long double fdim(long double, long double);
long double floor(long double);
long double fma(long double, long double, long double);
long double fmax(long double, long double);
long double fmin(long double, long double);
long double fmod(long double, long double);
long double frexp(long double, int*);
long double hypot(long double, long double);
int ilogb(long double);
long double ldexp(long double, int);
long double lgamma(long double);
long long llrint(long double);
long long llround(long double);
```

```
long double log(long double);
long double log10(long double);
long double log1p(long double);
long double log2(long double);
long double logb(long double);
long lrint(long double);
long lround(long double);
long double modf(long double, long double*);
long double nearbyint(long double);
long double nextafter(long double, long double);
long double nexttoward(long double, long double);
long double pow(long double, long double);
long double remainder(long double, long double);
long double remquo(long double, long double, int *);
long double rint(long double);
long double round(long double);
long double scalbln(long double, long);
long double scalbn(long double, int);
long double sin(long double);
long double sinh(long double);
long double sqrt(long double);
long double tan(long double);
long double tanh(long double);
long double tgamma(long double);
long double trunc(long double);
```

11 Moreover, there shall be additional overloads sufficient to ensure:

1. If any argument corresponding to a `double` parameter has type `long double`, then all arguments corresponding to `double` parameters are effectively cast to `long double`.
2. Otherwise, if any argument corresponding to a `double` parameter has type `double` or an integer type, then all arguments corresponding to `double` parameters are effectively cast to `double`.
3. Otherwise, all arguments corresponding to `double` parameters are effectively cast to `float`.

12 The templates defined in `<cmath>` replace the C99 macros with the same names. The templates have the following declarations:

```
namespace std {
        template <class T> bool signbit(T x);

        template <class T> int fpclassify(T x);

        template <class T> bool isfinite(T x);
        template <class T> bool isinf(T x);
        template <class T> bool isnan(T x);
        template <class T> bool isnormal(T x);

        template <class T> bool isgreater(T x, T y);
        template <class T> bool isgreaterequal(T x, T y);
        template <class T> bool isless(T x, T y);
        template <class T> bool islessequal(T x, T y);
        template <class T> bool islessgreater(T x, T y);
        template <class T> bool isunordered(T x, T y);
}
```

13 The templates behave the same as the C99 macros with corresponding names defined in C99 7.12.3, Classification macros, and C99 7.12.14, Comparison macros.

See also: ISO C 7.5, 7.10.2, 7.10.6.

27 Input/output library [input.output]

1 This Clause describes components that C++ programs may use to perform input/output operations.

2 The following subclauses describe requirements for stream parameters, and components for forward declarations of iostreams, predefined iostreams objects, base iostreams classes, stream buffering, stream formatting and manipulators, string streams, and file streams, as summarized in Table 99.

Table 99 — Input/output library summary

Subclause		Header(s)
27.1	Requirements	
27.2	Forward declarations	`<iosfwd>`
27.3	Standard iostream objects	`<iostream>`
27.4	Iostreams base classes	`<ios>`
27.5	Stream buffers	`<streambuf>`

27.6	Formatting and manipulators	`<istream>`
		`<ostream>`
		`<iomanip>`

27.7	String streams	`<sstream>`

27.8	File streams	`<fstream>`
		`<cstdio>`
		`<cinttypes>`

27.1 Iostreams requirements [iostreams.requirements]

27.1.1 Imbue Limitations [iostream.limits.imbue]

1 No function described in Clause 27 except for `ios_base::imbue` and `basic_filebuf::pubimbue` causes any instance of `basic_ios::imbue` or `basic_streambuf::imbue` to be called. If any user function called from a function declared in Clause 27 or as an overriding virtual function of any class declared in Clause 27 calls `imbue`, the behavior is undefined.

27.1.2 Positioning Type Limitations [iostreams.limits.pos]

1 The classes of Clause 27 with template arguments `charT` and `traits` behave as described if `traits::pos_type` and `traits::off_type` are `streampos` and `streamoff` respectively. Except as noted explicitly below, their behavior when `traits::pos_type` and `traits::off_type` are other types is implementation-defined.

27.1.3 Thread safety [iostreams.threadsafety]

1 Concurrent access to a stream object (27.7, 27.8), stream buffer object (27.5), or C Library stream (27.8.2) by multiple threads may result in a data race (1.10) unless otherwise specified (27.3). [*Note:* data races

result in undefined behavior (1.10). — *end note*]

27.2 Forward declarations [iostream.forward]

Header <iosfwd> synopsis

```
namespace std {
  template<class charT> class char_traits;
  template<> class char_traits<char>;
  template<> class char_traits<wchar_t>;

  template<class T> class allocator;

  template <class charT, class traits = char_traits<charT> >
    class basic_ios;
  template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
  template <class charT, class traits = char_traits<charT> >
    class basic_istream;
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
  template <class charT, class traits = char_traits<charT> >
    class basic_iostream;

  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_stringbuf;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_istringstream;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_ostringstream;
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
    class basic_stringstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_filebuf;
  template <class charT, class traits = char_traits<charT> >
    class basic_ifstream;
  template <class charT, class traits = char_traits<charT> >
    class basic_ofstream;
  template <class charT, class traits = char_traits<charT> >
    class basic_fstream;

  template <class charT, class traits = char_traits<charT> >
    class istreambuf_iterator;
  template <class charT, class traits = char_traits<charT> >
    class ostreambuf_iterator;

  typedef basic_ios<char>       ios;
  typedef basic_ios<wchar_t>    wios;

  typedef basic_streambuf<char> streambuf;
  typedef basic_istream<char>   istream;
```

```
  typedef basic_ostream<char>   ostream;
  typedef basic_iostream<char>  iostream;

  typedef basic_stringbuf<char>     stringbuf;
  typedef basic_istringstream<char> istringstream;
  typedef basic_ostringstream<char> ostringstream;
  typedef basic_stringstream<char>  stringstream;

  typedef basic_filebuf<char>  filebuf;
  typedef basic_ifstream<char> ifstream;
  typedef basic_ofstream<char> ofstream;
  typedef basic_fstream<char>  fstream;

  typedef basic_streambuf<wchar_t> wstreambuf;
  typedef basic_istream<wchar_t>   wistream;
  typedef basic_ostream<wchar_t>   wostream;
  typedef basic_iostream<wchar_t>  wiostream;

  typedef basic_stringbuf<wchar_t>     wstringbuf;
  typedef basic_istringstream<wchar_t> wistringstream;
  typedef basic_ostringstream<wchar_t> wostringstream;
  typedef basic_stringstream<wchar_t>  wstringstream;

  typedef basic_filebuf<wchar_t>  wfilebuf;
  typedef basic_ifstream<wchar_t> wifstream;
  typedef basic_ofstream<wchar_t> wofstream;
  typedef basic_fstream<wchar_t>  wfstream;

  template <class state> class fpos;
  typedef fpos<char_traits<char>::state_type>    streampos;
  typedef fpos<char_traits<wchar_t>::state_type> wstreampos;
}
```

1 Default template arguments are described as appearing both in `<iosfwd>` and in the synopsis of other headers but it is well-formed to include both `<iosfwd>` and one or more of the other headers.[289]

2 [*Note:* The class template specialization `basic_ios<charT,traits>` serves as a virtual base class for the class templates `basic_istream`, `basic_ostream`, and class templates derived from them. `basic_iostream` is a class template derived from both `basic_istream<charT,traits>` and `basic_ostream<charT,traits>`.

3 The class template specialization `basic_streambuf<charT,traits>` serves as a base class for template classes `basic_stringbuf` and `basic_filebuf`.

4 The class template specialization `basic_istream<charT,traits>` serves as a base class for template classes `basic_istringstream` and `basic_ifstream`.

5 The class template specialization `basic_ostream<charT,traits>` serves as a base class for template classes `basic_ostringstream` and `basic_ofstream`.

6 The class template specialization `basic_iostream<charT,traits>` serves as a base class for template classes `basic_stringstream` and `basic_fstream`.

7 Other typedefs define instances of class templates specialized for `char` or `wchar_t` types.

289) It is the implementation's responsibility to implement headers so that including `<iosfwd>` and other headers does not violate the rules about multiple occurences of default arguments.

8 Specializations of the class template `fpos` are used for specifying file position information.

9 The types `streampos` and `wstreampos` are used for positioning streams specialized on `char` and `wchar_t` respectively.

10 This synopsis suggests a circularity between `streampos` and `char_traits<char>`. An implementation can avoid this circularity by substituting equivalent types. One way to do this might be

```
template<class stateT> class fpos { ... };       // depends on nothing
typedef ... _STATE;              // implementation private declaration of stateT

typedef fpos<_STATE> streampos;

template<> struct char_traits<char> {
  typedef streampos
  pos_type;
}
```

— *end note*]

27.3 Standard iostream objects [iostream.objects]

Header <iostream> synopsis

```
#include <ios>
#include <streambuf>
#include <istream>
#include <ostream>

namespace std {
  extern istream cin;
  extern ostream cout;
  extern ostream cerr;
  extern ostream clog;

  extern wistream wcin;
  extern wostream wcout;
  extern wostream wcerr;
  extern wostream wclog;
}
```

1 The header `<iostream>` declares objects that associate objects with the standard C streams provided for by the functions declared in `<cstdio>` (27.8.2), and includes all the headers necessary to use these objects.

2 The objects are constructed and the associations are established at some time prior to or during the first time an object of class `ios_base::Init` is constructed, and in any case before the body of `main` begins execution.[290] The objects are *not* destroyed during program execution.[291] If a translation unit includes `<iostream>` or explicitly constructs an `ios_base::Init` object, these stream objects shall be constructed before dynamic initialization of non-local objects defined later in that translation unit.

3 Mixing operations on corresponding wide- and narrow-character streams follows the same semantics as mixing such operations on `FILE`s, as specified in Amendment 1 of the ISO C standard.

290) If it is possible for them to do so, implementations are encouraged to initialize the objects earlier than required.

291) Constructors and destructors for static objects can access these objects to read input from `stdin` or write output to `stdout` or `stderr`.

4 Concurrent access to a synchronized (27.4.2.4) standard iostream object's formatted and unformatted input (27.6.1.1) and output (27.6.2.1) functions or a standard C stream by multiple threads shall not result in a data race (1.10). [*Note:* users must still synchronize concurrent use of these objects and streams by multiple threads if they wish to avoid interleaved characters. — *end note*]

27.3.1 Narrow stream objects [narrow.stream.objects]

`istream cin;`

1 The object `cin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

2 After the object `cin` is initialized, `cin.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.4.4.1).

`ostream cout;`

3 The object `cout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.8.2).

`ostream cerr;`

4 The object `cerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

5 After the object `cerr` is initialized, `cerr.flags() & unitbuf` is nonzero and `cerr.tie()` returns `&cout`. Its state is otherwise the same as required for `basic_ios<char>::init` (27.4.4.1).

`ostream clog;`

6 The object `clog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

27.3.2 Wide stream objects [wide.stream.objects]

`wistream wcin;`

1 The object `wcin` controls input from a stream buffer associated with the object `stdin`, declared in `<cstdio>`.

2 After the object `wcin` is initialized, `wcin.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (27.4.4.1).

`wostream wcout;`

3 The object `wcout` controls output to a stream buffer associated with the object `stdout`, declared in `<cstdio>` (27.8.2).

`wostream wcerr;`

4 The object `wcerr` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

5 After the object `wcerr` is initialized, `wcerr.flags() & unitbuf` is nonzero and `wcerr.tie()` returns `&wcout`. Its state is otherwise the same as required for `basic_ios<wchar_t>::init` (27.4.4.1).

`wostream wclog;`

6 The object `wclog` controls output to a stream buffer associated with the object `stderr`, declared in `<cstdio>` (27.8.2).

27.4 Iostreams base classes [iostreams.base]

Header `<ios>` synopsis

```
#include <iosfwd>

namespace std {
  typedef OFF_T streamoff;
  typedef SZ_T streamsize;
  template <class stateT> class fpos;

  class ios_base;
  template <class charT, class traits = char_traits<charT> >
    class basic_ios;

  // 27.4.5, manipulators:
  ios_base& boolalpha  (ios_base& str);
  ios_base& noboolalpha(ios_base& str);

  ios_base& showbase   (ios_base& str);
  ios_base& noshowbase (ios_base& str);

  ios_base& showpoint  (ios_base& str);
  ios_base& noshowpoint(ios_base& str);

  ios_base& showpos    (ios_base& str);
  ios_base& noshowpos  (ios_base& str);

  ios_base& skipws     (ios_base& str);
  ios_base& noskipws   (ios_base& str);

  ios_base& uppercase  (ios_base& str);
  ios_base& nouppercase(ios_base& str);

  ios_base& unitbuf    (ios_base& str);
  ios_base& nounitbuf  (ios_base& str);

  // 27.4.5.2 adjustfield:
  ios_base& internal   (ios_base& str);
  ios_base& left       (ios_base& str);
  ios_base& right      (ios_base& str);

  // 27.4.5.3 basefield:
  ios_base& dec        (ios_base& str);
  ios_base& hex        (ios_base& str);
  ios_base& oct        (ios_base& str);

  // 27.4.5.4 floatfield:
  ios_base& fixed      (ios_base& str);
  ios_base& scientific (ios_base& str);
  ios_base& hexfloat   (ios_base& str);
  ios_base& defaultfloat(ios_base& str);
```

```
    // 27.4.5.5 error reporting:
    enum class io_errc {
      stream = 1
    };

    concept_map ErrorCodeEnum<io_errc> { };
    error_code make_error_code(io_errc e);
    error_condition make_error_condition(io_errc e);
    storage-class-specifier const error_category& iostream_category;
  }
```

27.4.1 Types [stream.types]

```
typedef OFF_T  streamoff;
```

1 The type `streamoff` is an implementation-defined type that satisfies the requirements of 27.4.3.2.

```
typedef SZ_T  streamsize;
```

2 The type `streamsize` is a synonym for one of the signed basic integral types. It is used to represent the number of characters transferred in an I/O operation, or the size of I/O buffers.[292]

27.4.2 Class `ios_base` [ios.base]

```
namespace std {
  class ios_base {
  public:
    class failure;

    typedef T1 fmtflags;
    static const fmtflags boolalpha;
    static const fmtflags dec;
    static const fmtflags fixed;
    static const fmtflags hex;
    static const fmtflags internal;
    static const fmtflags left;
    static const fmtflags oct;
    static const fmtflags right;
    static const fmtflags scientific;
    static const fmtflags showbase;
    static const fmtflags showpoint;
    static const fmtflags showpos;
    static const fmtflags skipws;
    static const fmtflags unitbuf;
    static const fmtflags uppercase;
    static const fmtflags adjustfield;
    static const fmtflags basefield;
    static const fmtflags floatfield;

    typedef T2 iostate;
    static const iostate badbit;
```

292) `streamsize` is used in most places where ISO C would use `size_t`. Most of the uses of `streamsize` could use `size_t`, except for the `strstreambuf` constructors, which require negative values. It should probably be the signed type corresponding to `size_t` (which is what Posix.2 calls `ssize_t`).

```
  static const iostate eofbit;
  static const iostate failbit;
  static const iostate goodbit;

  typedef T3 openmode;
  static const openmode app;
  static const openmode ate;
  static const openmode binary;
  static const openmode in;
  static const openmode out;
  static const openmode trunc;

  typedef T4 seekdir;
  static const seekdir beg;
  static const seekdir cur;
  static const seekdir end;

  class Init;

  // 27.4.2.2 fmtflags state:
  fmtflags flags() const;
  fmtflags flags(fmtflags fmtfl);
  fmtflags setf(fmtflags fmtfl);
  fmtflags setf(fmtflags fmtfl, fmtflags mask);
  void unsetf(fmtflags mask);

  streamsize precision() const;
  streamsize precision(streamsize prec);
  streamsize width() const;
  streamsize width(streamsize wide);

  // 27.4.2.3 locales:
  locale imbue(const locale& loc);
  locale getloc() const;

  // 27.4.2.5 storage:
  static int xalloc();
  long&  iword(int index);
  void*& pword(int index);

  // destructor
  virtual ~ios_base();

  // 27.4.2.6 callbacks;
  enum event { erase_event, imbue_event, copyfmt_event };
  typedef void (*event_callback)(event, ios_base&, int index);
  void register_callback(event_callback fn, int index);

  ios_base(const ios_base&) = delete;
  ios_base& operator=(const ios_base&) = delete;

  static bool sync_with_stdio(bool sync = true);

protected:
  ios_base();
```

```
  private:
    // static int index;                    exposition only
    // long* iarray;                        exposition only
    // void** parray;                       exposition only
  };
}
```

1 `ios_base` defines several member types:

— a class `failure` derived from `exception`;

— a class `Init`;

— three bitmask types, `fmtflags`, `iostate`, and `openmode`;

— an enumerated type, `seekdir`.

2 It maintains several kinds of data:

— state information that reflects the integrity of the stream buffer;

— control information that influences how to interpret (format) input sequences and how to generate (format) output sequences;

— additional information that is stored by the program for its private use.

3 [*Note:* For the sake of exposition, the maintained data is presented here as:

— `static int index`, specifies the next available unique index for the integer or pointer arrays maintained for the private use of the program, initialized to an unspecified value;

— `long* iarray`, points to the first element of an arbitrary-length `long` array maintained for the private use of the program;

— `void** parray`, points to the first element of an arbitrary-length pointer array maintained for the private use of the program. — *end note*]

27.4.2.1 Types [ios.types]

27.4.2.1.1 Class `ios_base::failure` [ios::failure]

```
namespace std {
  class ios_base::failure : public system_error {
  public:
    explicit failure(const string& msg, const error_code& ec = io_errc::stream);
    explicit failure(const char* msg, const error_code& ec = io_errc::stream);
    virtual const char* what() const throw();
  };
}
```

1 The class `failure` defines the base class for the types of all objects thrown as exceptions, by functions in the iostreams library, to report errors detected during stream buffer operations.

2 When throwing `ios_base::failure` exceptions, implementations should provide values of `ec` that identify the specific reason for the failure. [*Note:* Errors arising from the operating system would typically be reported as `system_category` errors with an error value of the error number reported by the operating system. Errors arising from within the stream library would typically be reported as `error_code(io_errc::stream, iostream_category)`. — *end note*]

```
explicit failure(const string& msg, , const error_code& ec = io_errc::stream);
```

3 *Effects:* Constructs an object of class `failure`.

4 *Postcondition:* `code() == ec` and `strcmp(what(), msg.c_str()) == 0`

```
explicit failure(const char* msg, const error_code& ec = io_errc::stream);
```

5 *Effects:* Constructs an object of class `failure`.

6 *Postcondition:* `code() == ec` and `strcmp(what(), msg) == 0`

```
const char* what() const;
```

7 *Returns:* The message `msg` with which the exception was created.

27.4.2.1.2 Type `ios_base::fmtflags` [ios::fmtflags]

```
typedef T1  fmtflags;
```

1 The type `fmtflags` is a bitmask type (17.5.3.2.3). Setting its elements has the effects indicated in Table 100.

Table 100 — `fmtflags` effects

Element	Effect(s) if set
`boolalpha`	insert and extract `bool` type in alphabetic format
`dec`	converts integer input or generates integer output in decimal base
`fixed`	generate floating-point output in fixed-point notation
`hex`	converts integer input or generates integer output in hexadecimal base
`internal`	adds fill characters at a designated internal point in certain generated output, or identical to `right` if no such point is designated
`left`	adds fill characters on the right (final positions) of certain generated output
`oct`	converts integer input or generates integer output in octal base
`right`	adds fill characters on the left (initial positions) of certain generated output
`scientific`	generates floating-point output in scientific notation
`showbase`	generates a prefix indicating the numeric base of generated integer output
`showpoint`	generates a decimal-point character unconditionally in generated floating-point output
`showpos`	generates a `+` sign in non-negative generated numeric output
`skipws`	skips leading whitespace before certain input operations
`unitbuf`	flushes output after each output operation
`uppercase`	replaces certain lowercase letters with their uppercase equivalents in generated output

2 Type `fmtflags` also defines the constants indicated in Table 101.

Table 101 — `fmtflags` constants

Constant	Allowable values
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

27.4.2.1.3 Type `ios_base::iostate` [ios::iostate]

```
typedef T2  iostate;
```

1 The type `iostate` is a bitmask type (17.5.3.2.3) that contains the elements indicated in Table 102.

Table 102 — `iostate` effects

Element	Effect(s) if set
`badbit`	indicates a loss of integrity in an input or output sequence (such as an irrecoverable read error from a file);
`eofbit`	indicates that an input operation reached the end of an input sequence;
`failbit`	indicates that an input operation failed to read the expected characters, or that an output operation failed to generate the desired characters.

2 Type `iostate` also defines the constant:

— `goodbit`, the value zero.

27.4.2.1.4 Type `ios_base::openmode` [ios::openmode]

```
typedef T3  openmode;
```

1 The type `openmode` is a bitmask type (17.5.3.2.3). It contains the elements indicated in Table 103.

Table 103 — `openmode` effects

Element	Effect(s) if set
`app`	seek to end before each write
`ate`	open and seek to end immediately after opening
`binary`	perform input and output in binary mode (as opposed to text mode)
`in`	open for input
`out`	open for output
`trunc`	truncate an existing stream when opening

27.4.2.1.5 Type `ios_base::seekdir` [ios::seekdir]

```
typedef T4  seekdir;
```

1 The type `seekdir` is an enumerated type (17.5.3.2.2) that contains the elements indicated in Table 104.

Table 104 — `seekdir` effects

Element	Meaning
`beg`	request a seek (for subsequent input or output) relative to the beginning of the stream
`cur`	request a seek relative to the current position within the sequence
`end`	request a seek relative to the current end of the sequence

27.4.2.1.6 Class `ios_base::Init` [ios::Init]

```
namespace std {
  class ios_base::Init {
  public:
    Init();
   ~Init();
  private:
    // static int init_cnt;                                    exposition only
  };
}
```

1 The class `Init` describes an object whose construction ensures the construction of the eight objects declared in `<iostream>` (27.3) that associate file stream buffers with the standard C streams provided for by the functions declared in `<cstdio>` (27.8.2).

2 For the sake of exposition, the maintained data is presented here as:

— `static int init_cnt`, counts the number of constructor and destructor calls for class `Init`, initialized to zero.

`Init();`

3 *Effects:* Constructs an object of class `Init`. If `init_cnt` is zero, the function stores the value one in `init_cnt`, then constructs and initializes the objects `cin`, `cout`, `cerr`, `clog` (27.3.1), `wcin`, `wcout`, `wcerr`, and `wclog` (27.3.2). In any case, the function then adds one to the value stored in `init_cnt`.

`~Init();`

4 *Effects:* Destroys an object of class `Init`. The function subtracts one from the value stored in `init_cnt` and, if the resulting stored value is one, calls `cout.flush()`, `cerr.flush()`, `clog.flush()`, `wcout.flush()`, `wcerr.flush()`, `wclog.flush()`.

27.4.2.2 `ios_base` state functions [fmtflags.state]

`fmtflags flags() const;`

1 *Returns:* The format control information for both input and output.

`fmtflags flags(fmtflags fmtfl);`

2 *Postcondition:* `fmtfl == flags()`.

3 *Returns:* The previous value of `flags()`.

`fmtflags setf(fmtflags fmtfl);`

4 *Effects:* Sets `fmtfl` in `flags()`.

5 *Returns:* The previous value of `flags()`.

`fmtflags setf(fmtflags fmtfl, fmtflags mask);`

6 *Effects:* Clears `mask` in `flags()`, sets `fmtfl & mask` in `flags()`.

7 *Returns:* The previous value of `flags()`.

`void unsetf(fmtflags mask);`

8 *Effects:* Clears `mask` in `flags()`.

`streamsize precision() const;`

9 *Returns:* The precision to generate on certain output conversions.

```
streamsize precision(streamsize prec);
```

10 *Postcondition:* `prec == precision()`.

11 *Returns:* The previous value of `precision()`.

```
streamsize width() const;
```

12 *Returns:* The minimum field width (number of characters) to generate on certain output conversions.

```
streamsize width(streamsize wide);
```

13 *Postcondition:* `wide == width()`.

14 *Returns:* The previous value of `width()`.

27.4.2.3 `ios_base` functions [ios.base.locales]

```
locale imbue(const locale& loc);
```

1 *Effects:* Calls each registered callback pair `(fn,index)` (27.4.2.6) as `(*fn)(imbue_event,*this,index)` at such a time that a call to `ios_base::getloc()` from within `fn` returns the new locale value `loc`.

2 *Returns:* The previous value of `getloc()`.

3 *Postcondition:* `loc == getloc()`.

```
locale getloc() const;
```

4 *Returns:* If no locale has been imbued, a copy of the global C++ locale, `locale()`, in effect at the time of construction. Otherwise, returns the imbued locale, to be used to perform locale-dependent input and output operations.

27.4.2.4 `ios_base` static members [ios.members.static]

```
bool sync_with_stdio(bool sync = true);
```

1 *Returns:* `true` if the previous state of the standard iostream objects (27.3) was synchronized and otherwise returns `false`. The first time it is called, the function returns `true`.

2 *Effects:* If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard streams to operate independently of the standard C streams.

3 When a standard iostream object `str` is *synchronized* with a standard stdio stream `f`, the effect of inserting a character `c` by

```
fputc(f, c);
```

is the same as the effect of

```
str.rdbuf()->sputc(c);
```

for any sequences of characters; the effect of extracting a character `c` by

```
c = fgetc(f);
```

is the same as the effect of

```
c = str.rdbuf()->sbumpc(c);
```

for any sequences of characters; and the effect of pushing back a character `c` by

```
ungetc(c, f);
```

is the same as the effect of

```
str.rdbuf()->sputbackc(c);
```

for any sequence of characters.[293]

27.4.2.5 `ios_base` storage functions [ios.base.storage]

```
static int xalloc();
```

1 *Returns:* `index ++`.

```
long& iword(int idx);
```

2 *Effects:* If `iarray` is a null pointer, allocates an array of `long` of unspecified size and stores a pointer to its first element in `iarray`. The function then extends the array pointed at by `iarray` as necessary to include the element `iarray[idx]`. Each newly allocated element of the array is initialized to zero. The reference returned is invalid after any other operations on the object.[294] However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `iword` with the same index yields another reference to the same value. If the function fails[295] and `*this` is a base subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

3 *Returns:* On success `iarray[idx]`. On failure, a valid `long&` initialized to 0.

```
void* & pword(int idx);
```

4 *Effects:* If `parray` is a null pointer, allocates an array of pointers to `void` of unspecified size and stores a pointer to its first element in `parray`. The function then extends the array pointed at by `parray` as necessary to include the element `parray[idx]`. Each newly allocated element of the array is initialized to a null pointer. The reference returned is invalid after any other operations on the object. However, the value of the storage referred to is retained, so that until the next call to `copyfmt`, calling `pword` with the same index yields another reference to the same value. If the function fails[296] and `*this` is a base subobject of a `basic_ios<>` object or subobject, the effect is equivalent to calling `basic_ios<>::setstate(badbit)` on the derived object (which may throw `failure`).

5 *Returns:* On success `parray[idx]`. On failure a valid `void*&` initialized to 0.

6 *Remarks:* After a subsequent call to `pword(int)` for the same object, the earlier return value may no longer be valid.

293) This implies that operations on a standard iostream object can be mixed arbitrarily with operations on the corresponding stdio stream. In practical terms, synchronization usually means that a standard iostream object and a standard stdio object share a buffer.

294) An implementation is free to implement both the integer array pointed at by `iarray` and the pointer array pointed at by `parray` as sparse data structures, possibly with a one-element cache for each.

295) for example, because it cannot allocate space.

296) for example, because it cannot allocate space.

27.4.2.6 `ios_base` callbacks [ios.base.callback]

```
void register_callback(event_callback fn, int index);
```

1 *Effects:* Registers the pair `(fn,index)` such that during calls to `imbue()` (27.4.2.3), `copyfmt()`, or `~ios_base()` (27.4.2.7), the function `fn` is called with argument `index`. Functions registered are called when an event occurs, in opposite order of registration. Functions registered while a callback function is active are not called until the next event.

2 *Requires:* The function `fn` shall not throw exceptions.

Remarks: Identical pairs are not merged. A function registered twice will be called twice.

27.4.2.7 `ios_base` constructors/destructor [ios.base.cons]

```
ios_base();
```

1 *Effects:* Each `ios_base` member has an indeterminate value after construction. These members shall be initialized by calling `basic_ios::init`. If an `ios_base` object is destroyed before these initializations have taken place, the behavior is undefined.

```
~ios_base()
```

2 *Effects:* Destroys an object of class `ios_base`. Calls each registered callback pair `(fn, index)` (27.4.2.6) as `(*fn)(erase_event, *this, index)` at such time that any `ios_base` member function called from within `fn` has well defined results.

27.4.3 Class template `fpos` [fpos]

```
namespace std {
  template <class stateT> class fpos {
  public:
    // 27.4.3.1 Members
    stateT state() const;
    void state(stateT);
  private;
    // stateT st;                                   exposition only
  };
}
```

27.4.3.1 `fpos` Members [fpos.members]

```
void state(stateT s);
```

1 *Effects:* Assign `s` to `st`.

```
stateT state() const;
```

2 *Returns:* Current value of `st`.

27.4.3.2 `fpos` requirements [fpos.operations]

1 Operations specified in Table 105 are permitted. In that table,

— `P` refers to an instance of `fpos`,

— `p` and `q` refer to values of type `P`,

— `O` refers to type `streamoff`,

— `o` refers to a value of type `streamoff`,

— `sz` refers to a value of type `streamsize` and

— `i` refers to a value of type `int`.

Table 105 — Position type requirements

Expression	Return type	Operational semantics	Assertion/note pre-/post-condition
`P(i)`			`p == P(i)` note: a destructor is assumed.
`P p(i);` `P p = i;`			post: `p == P(i)`.
`P(o)`	`fpos`	converts from `offset`	
`O(p)`	`OFF_T`	converts to `offset`	`P(O(p)) == p`
`p == q`	convertible to `bool`		`==` is an equivalence relation
`p != q`	convertible to `bool`	`!(p == q)`	
`q = p + o` `p += o`	`fpos`	`+` offset	`q - o == p`
`q = p - o` `p -= o`	`fpos`	`-` offset	`q + o == p`
`o = p - q`	`OFF_T`	distance	`q + o == p`
`streamsize(o)`	`streamsize`	converts	`streamsize(O(sz)) == sz`
`O(sz)`	`OFF_T`	converts	`streamsize(O(sz)) == sz`

2 [*Note:* Every implementation is required to supply overloaded operators on `fpos` objects to satisfy the requirements of 27.4.3.2. It is unspecified whether these operators are members of `fpos`, global operators, or provided in some other way. — *end note*]

3 Stream operations that return a value of type `traits::pos_type` return `P(O(-1))` as an invalid value to signal an error. If this value is used as an argument to any `istream`, `ostream`, or `streambuf` member that accepts a value of type `traits::pos_type` then the behavior of that function is undefined.

27.4.4 Class template `basic_ios` [ios]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ios : public ios_base {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    operator unspecified-bool-type() const;
    bool operator!() const;
    iostate rdstate() const;
```

```
    void clear(iostate state = goodbit);
    void setstate(iostate state);
    bool good() const;
    bool eof()  const;
    bool fail() const;
    bool bad()  const;

    iostate exceptions() const;
    void exceptions(iostate except);

    // 27.4.4.1 Constructor/destructor:
    explicit basic_ios(basic_streambuf<charT,traits>* sb);
    virtual ~basic_ios();

    // 27.4.4.2 Members:
    basic_ostream<charT,traits>* tie() const;
    basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);

    basic_streambuf<charT,traits>* rdbuf() const;
    basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);

    basic_ios& copyfmt(const basic_ios& rhs);

    char_type fill() const;
    char_type fill(char_type ch);

    locale imbue(const locale& loc);

    char     narrow(char_type c, char dfault) const;
    char_type widen(char c) const;

    basic_ios(const basic_ios& ) = delete;
    basic_ios& operator=(const basic_ios&) = delete;

  protected:
    basic_ios();
    void init(basic_streambuf<charT,traits>* sb);
    void move(basic_ios&& rhs);
    void swap(basic_ios&& rhs);
    void set_rdbuf(basic_streambuf<charT, traits>* sb);

  };
}
```

27.4.4.1 `basic_ios` constructors [basic.ios.cons]

```
explicit basic_ios(basic_streambuf<charT,traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_ios`, assigning initial values to its member objects by calling `init(sb)`.

```
basic_ios();
```

2 *Effects:* Constructs an object of class `basic_ios` (27.4.2.7) leaving its member objects uninitialized. The object shall be initialized by calling its `init` member function. If it is destroyed before it has been initialized the behavior is undefined.

```
~basic_ios();
```

3 *Remarks:* The destructor does not destroy `rdbuf()`.

```
void init(basic_streambuf<charT,traits>* sb);
```

Postconditions: The postconditions of this function are indicated in Table 106.

Table 106 — `basic_ios::init()` effects

Element	Value
`rdbuf()`	`sb`
`tie()`	`0`
`rdstate()`	`goodbit` if `sb` is not a null pointer, otherwise `badbit`.
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`width()`	`0`
`precision()`	`6`
`fill()`	`widen(' ');`
`getloc()`	a copy of the value returned by `locale()`
iarray	a null pointer
parray	a null pointer

27.4.4.2 Member functions [basic.ios.members]

```
basic_ostream<charT,traits>* tie() const;
```

1 *Returns:* An output sequence that is *tied* to (synchronized with) the sequence controlled by the stream buffer.

```
basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);
```

2 *Postcondition:* `tiestr == tie()`.

3 *Returns:* The previous value of `tie()`.

```
basic_streambuf<charT,traits>* rdbuf() const;
```

4 *Returns:* A pointer to the `streambuf` associated with the stream.

```
basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);
```

5 *Postcondition:* `sb == rdbuf()`.

6 *Effects:* Calls `clear()`.

7 *Returns:* The previous value of `rdbuf()`.

```
locale imbue(const locale& loc);
```

8 *Effects:* Calls `ios_base::imbue(loc)` (27.4.2.3) and if `rdbuf()!=0` then `rdbuf()->pubimbue(loc)` (27.5.2.2.1).

9 *Returns:* The prior value of `ios_base::imbue()`.

```
char narrow(char_type c, char dfault) const;
```

10 *Returns:* `use_facet< ctype<char_type> >(getloc()).narrow(c,dfault)`

```
char_type widen(char c) const;
```

11 *Returns:* `use_facet< ctype<char_type> >(getloc()).widen(c)`

```
char_type fill() const;
```

12 *Returns:* The character used to pad (fill) an output conversion to the specified field width.

```
char_type fill(char_type fillch);
```

13 *Postcondition:* `traits::eq(fillch, fill())`

14 *Returns:* The previous value of `fill()`.

```
basic_ios& copyfmt(const basic_ios& rhs);
```

15 *Effects:* If `(this == &rhs)` does nothing. Otherwise assigns to the member objects of `*this` the corresponding member objects of `rhs`, except that:

— `rdstate()` and `rdbuf()` are left unchanged;

— `exceptions()` is altered last by calling `exceptions(rhs.except)`.

— The contents of arrays pointed at by `pword` and `iword` are copied not the pointers themselves.[297]

16 If any newly stored pointer values in `*this` point at objects stored outside the object `rhs`, and those objects are destroyed when `rhs` is destroyed, the newly stored pointer values are altered to point at newly constructed copies of the objects.

17 Before copying any parts of `rhs`, calls each registered callback pair `(fn, index)` as `(*fn)(erase_event, *this, index)`. After all parts but `exceptions()` have been replaced, calls each callback pair that was copied from `rhs` as `(*fn)(copyfmt_event,*this,index)`.

18 *Remarks:* The second pass permits a copied `pword` value to be zeroed, or its referent deep copied or reference counted or have other special action taken.

19 *Returns:* `*this`.

```
void move(basic_ios&& rhs);
```

20 *Postconditions:* `*this` shall have the state that `rhs` had before the function call, except that `rdbuf()` shall return 0. `rhs` shall be in a valid but unspecified state, except that `rhs.rdbuf()` shall return the same value as it returned before the function call, and `rhs.tie()` shall return 0.

```
void swap(basic_ios&& rhs);
```

21 *Effects:* The states of `*this` and `rhs` shall be exchanged, except that `rdbuf()` shall return the same value as it returned before the function call, and `rhs.rdbuf()` shall return the same value as it returned before the function call.

22 *Throws:* Nothing.

```
void set_rdbuf(basic_streambuf<charT, traits>* sb);
```

23 *Effects:* Associates the `basic_streambuf` object pointed to by `sb` with this stream without calling `clear()`. *Postconditions:* `rdbuf() == sb`.

25 *Throws:* Nothing.

297) This suggests an infinite amount of copying, but the implementation can keep track of the maximum element of the arrays that is non-zero.

27.4.4.3 `basic_ios` flags functions [iostate.flags]

```
operator unspecified-bool-type() const;
```

1 *Returns:* If `fail()` then a value that will evaluate false in a boolean context; otherwise a value that will evaluate true in a boolean context. The value type returned shall not be convertible to int.

[*Note:* This conversion can be used in contexts where a `bool` is expected (e.g., an `if` condition); however, implicit conversions (e.g., to `int`) that can occur with `bool` are not allowed, eliminating some sources of user error. One possible implementation choice for this type is pointer-to-member. — *end note*]

```
bool operator!() const;
```

2 *Returns:* `fail()`.

```
iostate rdstate() const;
```

3 *Returns:* The error state of the stream buffer.

```
void clear(iostate state = goodbit);
```

4 *Postcondition:* If `rdbuf()!=0` then `state == rdstate()`; otherwise `rdstate()==(state | ios_base::badbit)`.

5 *Effects:* If `((state | (rdbuf() ? goodbit : badbit)) & exceptions()) == 0`, returns. Otherwise, the function throws an object `fail` of class `basic_ios::failure` (27.4.2.1.1), constructed with implementation-defined argument values.

```
void setstate(iostate state);
```

6 *Effects:* Calls `clear(rdstate() | state)` (which may throw `basic_ios::failure` (27.4.2.1.1)).

```
bool good() const;
```

7 *Returns:* `rdstate() == 0`

```
bool eof() const;
```

8 *Returns:* `true` if `eofbit` is set in `rdstate()`.

```
bool fail() const;
```

9 *Returns:* `true` if `failbit` or `badbit` is set in `rdstate()`.[298]

```
bool bad() const;
```

10 *Returns:* `true` if `badbit` is set in `rdstate()`.

```
iostate exceptions() const;
```

11 *Returns:* A mask that determines what elements set in `rdstate()` cause exceptions to be thrown.

```
void exceptions(iostate except);
```

12 *Postcondition:* `except == exceptions()`.

13 *Effects:* Calls `clear(rdstate())`.

298) Checking `badbit` also for `fail()` is historical practice.

27.4.5 `ios_base` manipulators [std.ios.manip]

27.4.5.1 `fmtflags` manipulators [fmtflags.manip]

```
ios_base& boolalpha(ios_base& str);
```

1 *Effects:* Calls `str.setf(ios_base::boolalpha)`.

2 *Returns:* `str`.

```
ios_base& noboolalpha(ios_base& str);
```

3 *Effects:* Calls `str.unsetf(ios_base::boolalpha)`.

4 *Returns:* `str`.

```
ios_base& showbase(ios_base& str);
```

5 *Effects:* Calls `str.setf(ios_base::showbase)`.

6 *Returns:* `str`.

```
ios_base& noshowbase(ios_base& str);
```

7 *Effects:* Calls `str.unsetf(ios_base::showbase)`.

8 *Returns:* `str`.

```
ios_base& showpoint(ios_base& str);
```

9 *Effects:* Calls `str.setf(ios_base::showpoint)`.

10 *Returns:* `str`.

```
ios_base& noshowpoint(ios_base& str);
```

11 *Effects:* Calls `str.unsetf(ios_base::showpoint)`.

12 *Returns:* `str`.

```
ios_base& showpos(ios_base& str);
```

13 *Effects:* Calls `str.setf(ios_base::showpos)`.

14 *Returns:* `str`.

```
ios_base& noshowpos(ios_base& str);
```

15 *Effects:* Calls `str.unsetf(ios_base::showpos)`.

16 *Returns:* `str`.

```
ios_base& skipws(ios_base& str);
```

17 *Effects:* Calls `str.setf(ios_base::skipws)`.

18 *Returns:* `str`.

```
ios_base& noskipws(ios_base& str);
```

19 *Effects:* Calls `str.unsetf(ios_base::skipws)`.

20 *Returns:* `str`.

```
ios_base& uppercase(ios_base& str);
```

Effects: Calls `str.setf(ios_base::uppercase)`.

Returns: `str`.

```
ios_base& nouppercase(ios_base& str);
```

Effects: Calls `str.unsetf(ios_base::uppercase)`.

Returns: `str`.

```
ios_base& unitbuf(ios_base& str);
```

Effects: Calls `str.setf(ios_base::unitbuf)`.

Returns: `str`.

```
ios_base& nounitbuf(ios_base& str);
```

Effects: Calls `str.unsetf(ios_base::unitbuf)`.

Returns: `str`.

27.4.5.2 `adjustfield` manipulators [adjustfield.manip]

```
ios_base& internal(ios_base& str);
```

Effects: Calls `str.setf(ios_base::internal, ios_base::adjustfield)`.

Returns: `str`.

```
ios_base& left(ios_base& str);
```

Effects: Calls `str.setf(ios_base::left, ios_base::adjustfield)`.

Returns: `str`.

```
ios_base& right(ios_base& str);
```

Effects: Calls `str.setf(ios_base::right, ios_base::adjustfield)`.

Returns: `str`.

27.4.5.3 `basefield` manipulators [basefield.manip]

```
ios_base& dec(ios_base& str);
```

Effects: Calls `str.setf(ios_base::dec, ios_base::basefield)`.

Returns: `str`[299].

```
ios_base& hex(ios_base& str);
```

Effects: Calls `str.setf(ios_base::hex, ios_base::basefield)`.

Returns: `str`.

```
ios_base& oct(ios_base& str);
```

299) The function signature `dec(ios_base&)` can be called by the function signature `basic_ostream& stream::operator<<(ios_base& (*)(ios_base&))` to permit expressions of the form `cout <<dec` to change the format flags stored in `cout`.

Effects: Calls `str.setf(ios_base::oct, ios_base::basefield)`.

Returns: `str`.

27.4.5.4 `floatfield` manipulators [floatfield.manip]

```
ios_base& fixed(ios_base& str);
```

Effects: Calls `str.setf(ios_base::fixed, ios_base::floatfield)`.

Returns: `str`.

```
ios_base& scientific(ios_base& str);
```

Effects: Calls `str.setf(ios_base::scientific, ios_base::floatfield)`.

Returns: `str`.

```
ios_base& hexfloat(ios_base& str);
```

Effects: Calls `str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield)`.

Returns: `str`.

[*Note:* The more obvious use of `ios_base::hex` to specify hexadecimal floating-point format would change the meaning of existing well defined programs. C++2003 gives no meaning to the combination of `fixed` and `scientific`. — *end note*]

```
ios_base& defaultfloat(ios_base& str);
```

Effects: Calls `str.unsetf(ios_base::floatfield)`.

Returns: `str`.

27.4.5.5 Error reporting [error.reporting]

```
error_code make_error_code(io_errc e);
```

Returns: `error_code(static_cast<int>(e), iostream_category)`.

```
error_condition make_error_condition(io_errc e);
```

Returns: `error_condition(static_cast<int>(e), iostream_category)`.

storage-class-specifier `const error_category& iostream_category;`

The implementation shall initialize `iostream_category`. Its *storage-class-specifier* may be `static` or `extern`. It is unspecified whether initialization is static or dynamic (3.6.2). If initialization is dynamic, it shall occur before completion of the dynamic initialization of the first translation unit dynamically initialized that includes header `<system_error>`.

The object's `default_error_condition` and `equivalent` virtual functions shall behave as specified for the class `error_category`. The object's `name` virtual function shall return a pointer to the string `"iostream"`.

27.5 Stream buffers [stream.buffers]

Header `<streambuf>` synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_streambuf;
  typedef basic_streambuf<char>     streambuf;
  typedef basic_streambuf<wchar_t> wstreambuf;
}
```

1 The header <streambuf> defines types that control input from and output to *character* sequences.

27.5.1 Stream buffer requirements [streambuf.reqts]

1 Stream buffers can impose various constraints on the sequences they control. Some constraints are:

— The controlled input sequence can be not readable.

— The controlled output sequence can be not writable.

— The controlled sequences can be associated with the contents of other representations for character sequences, such as external files.

— The controlled sequences can support operations *directly* to or from associated sequences.

— The controlled sequences can impose limitations on how the program can read characters from a sequence, write characters to a sequence, put characters back into an input sequence, or alter the stream position.

2 Each sequence is characterized by three pointers which, if non-null, all point into the same charT array object. The array object represents, at any moment, a (sub)sequence of characters from the sequence. Operations performed on a sequence alter the values stored in these pointers, perform reads and writes directly to or from associated sequences, and alter "the stream position" and conversion state as needed to maintain this subsequence relationship. The three pointers are:

— the *beginning pointer,* or lowest element address in the array (called xbeg here);

— the *next pointer,* or next element address that is a current candidate for reading or writing (called xnext here);

— the *end pointer,* or first element address beyond the end of the array (called xend here).

3 The following semantic constraints shall always apply for any set of three pointers for a sequence, using the pointer names given immediately above:

— If xnext is not a null pointer, then xbeg and xend shall also be non-null pointers into the same charT array, as described above; otherwise, xbeg and xend shall also be null.

— If xnext is not a null pointer and xnext < xend for an output sequence, then a *write position* is available. In this case, *xnext shall be assignable as the next element to write (to put, or to store a character value, into the sequence).

— If xnext is not a null pointer and xbeg < xnext for an input sequence, then a *putback position* is available. In this case, xnext[-1] shall have a defined value and is the next (preceding) element to store a character that is put back into the input sequence.

— If xnext is not a null pointer and xnext < xend for an input sequence, then a *read position* is available. In this case, *xnext shall have a defined value and is the next element to read (to get, or to obtain a character value, from the sequence).

27.5.2 Class template `basic_streambuf<charT,traits>` [streambuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    virtual ~basic_streambuf();

    // 27.5.2.2.1 locales:
    locale   pubimbue(const locale& loc);
    locale   getloc() const;

    // 27.5.2.2.2 buffer and positioning:
    basic_streambuf<char_type,traits>*
       pubsetbuf(char_type* s, streamsize n);
    pos_type pubseekoff(off_type off, ios_base::seekdir way,
      ios_base::openmode which =
          ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
      ios_base::openmode which =
          ios_base::in | ios_base::out);
    int      pubsync();

    // Get and put areas:
    // 27.5.2.2.3 Get area:
    streamsize in_avail();
    int_type snextc();
    int_type sbumpc();
    int_type sgetc();
    streamsize sgetn(char_type* s, streamsize n);

    // 27.5.2.2.4 Putback:
    int_type sputbackc(char_type c);
    int_type sungetc();

    // 27.5.2.2.5 Put area:
    int_type   sputc(char_type c);
    streamsize sputn(const char_type* s, streamsize n);

  protected:
    basic_streambuf();
    basic_streambuf(const basic_streambuf& rhs);
    basic_streambuf& operator=(const basic_streambuf& rhs);

    void swap(basic_streambuf&& rhs);

    // 27.5.2.3.2 Get area:
```

```
    char_type* eback() const;
    char_type* gptr()  const;
    char_type* egptr() const;
    void       gbump(int n);
    void       setg(char_type* gbeg, char_type* gnext, char_type* gend);

    // 27.5.2.3.3 Put area:
    char_type* pbase() const;
    char_type* pptr() const;
    char_type* epptr() const;
    void       pbump(int n);
    void       setp(char_type* pbeg, char_type* pend);

    // 27.5.2.4 virtual functions:
    // 27.5.2.4.1 Locales:
    virtual void imbue(const locale& loc);

    // 27.5.2.4.2 Buffer management and positioning:
    virtual basic_streambuf<char_type,traits>*
         setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int      sync();

    // 27.5.2.4.3 Get area:
    virtual streamsize showmanyc();
    virtual streamsize xsgetn(char_type* s, streamsize n);
    virtual int_type   underflow();
    virtual int_type   uflow();

    // 27.5.2.4.4 Putback:
    virtual int_type   pbackfail(int_type c = traits::eof());

    // 27.5.2.4.5 Put area:
    virtual streamsize xsputn(const char_type* s, streamsize n);
    virtual int_type   overflow (int_type c = traits::eof());
  };
}
```

1 The class template `basic_streambuf<charT,traits>` serves as an abstract base class for deriving various *stream buffers* whose objects each control two *character sequences*:

— a character *input sequence*;

— a character *output sequence*.

2 [*Note:* This paragraph is intentionally empty. — *end note*]

3 [*Note:* This paragraph is intentionally empty. — *end note*]

27.5.2.1 `basic_streambuf` constructors [streambuf.cons]

```
basic_streambuf();
```

1 *Effects:* Constructs an object of class `basic_streambuf<charT,traits>` and initializes:[300]

- all its pointer member objects to null pointers,
- the `getloc()` member to a copy the global locale, `locale()`, at the time of construction.

2 *Remarks:* Once the `getloc()` member is initialized, results of calling locale member functions, and of members of facets so obtained, can safely be cached until the next time the member `imbue` is called.

```
basic_streambuf(const basic_streambuf& rhs);
```

3 *Effects:* Constructs a copy of `rhs`.

4 *Postconditions:*

- `eback() == rhs.eback()`
- `gptr() == rhs.gptr()`
- `egptr() == rhs.egptr()`
- `pbase() == rhs.pbase()`
- `pptr() == rhs.pptr()`
- `epptr() == rhs.epptr()`
- `getloc() == rhs.getloc()`

```
~basic_streambuf();
```

5 *Effects:* None.

27.5.2.2 `basic_streambuf` public member functions [streambuf.members]

27.5.2.2.1 Locales [streambuf.locales]

```
locale pubimbue(const locale& loc);
```

1 *Postcondition:* `loc == getloc()`.

2 *Effects:* Calls `imbue(loc)`.

3 *Returns:* Previous value of `getloc()`.

```
locale getloc() const;
```

4 *Returns:* If `pubimbue()` has ever been called, then the last value of `loc` supplied, otherwise the current global locale, `locale()`, in effect at the time of construction. If called after `pubimbue()` has been called but before `pubimbue` has returned (i.e. from within the call of `imbue()`) then it returns the previous value.

27.5.2.2.2 Buffer management and positioning [streambuf.buffer]

```
basic_streambuf<char_type,traits>* pubsetbuf(char_type* s, streamsize n);
```

1 *Returns:* `setbuf(s, n)`.

300) The default constructor is protected for class `basic_streambuf` to assure that only objects for classes derived from this class may be constructed.

```
pos_type pubseekoff(off_type off, ios_base::seekdir way,
                ios_base::openmode which = ios_base::in | ios_base::out);
```

2 *Returns:* `seekoff(off, way, which)`.

```
pos_type pubseekpos(pos_type sp,
                ios_base::openmode which = ios_base::in | ios_base::out);
```

3 *Returns:* `seekpos(sp, which)`.

```
int pubsync();
```

4 *Returns:* `sync()`.

27.5.2.2.3 Get area [streambuf.pub.get]

```
streamsize in_avail();
```

1 *Returns:* If a read position is available, returns `egptr() - gptr()`. Otherwise returns `showmanyc()` (27.5.2.4.3).

```
int_type snextc();
```

2 *Effects:* Calls `sbumpc()`.

3 *Returns:* if that function returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns `sgetc()`.

```
int_type sbumpc();
```

4 *Returns:* If the input sequence read position is not available, returns `uflow()`. Otherwise, returns `traits::to_int_type(*gptr())` and increments the next pointer for the input sequence.

```
int_type sgetc();
```

5 *Returns:* If the input sequence read position is not available, returns `underflow()`. Otherwise, returns `traits::to_int_type(*gptr())`.

```
streamsize sgetn(char_type* s, streamsize n);
```

6 *Returns:* `xsgetn(s, n)`.

27.5.2.2.4 Putback [streambuf.pub.pback]

```
int_type sputbackc(char_type c);
```

1 *Returns:* If the input sequence putback position is not available, or if `traits::eq(c,gptr()[-1])` is false, returns `pbackfail(traits::to_int_type(c))`. Otherwise, decrements the next pointer for the input sequence and returns `traits::to_int_type(*gptr())`.

```
int_type sungetc();
```

2 *Returns:* If the input sequence putback position is not available, returns `pbackfail()`. Otherwise, decrements the next pointer for the input sequence and returns `traits::to_int_type(*gptr())`.

27.5.2.2.5 Put area [streambuf.pub.put]

```
int_type sputc(char_type c);
```

1 *Returns:* If the output sequence write position is not available, returns `overflow(traits::to_int_type(c))`. Otherwise, stores `c` at the next pointer for the output sequence, increments the pointer, and returns `traits::to_int_type(c)`.

```
streamsize sputn(const char_type* s, streamsize n);
```

2 *Returns:* `xsputn(s,n)`.

27.5.2.3 `basic_streambuf` protected member functions [streambuf.protected]

27.5.2.3.1 Assignment [streambuf.assign]

```
basic_streambuf& operator=(const basic_streambuf& rhs);
```

1 *Effects:* Assigns the data members of `rhs` to `*this`.

2 *Postconditions:*

- `eback() == rhs.eback()`
- `gptr() == rhs.gptr()`
- `egptr() == rhs.egptr()`
- `pbase() == rhs.pbase()`
- `pptr() == rhs.pptr()`
- `epptr() == rhs.epptr()`
- `getloc() == rhs.getloc()`

3 *Returns:* `*this`.

```
void swap(basic_streambuf&& rhs);
```

4 *Effects:* Swaps the data members of `rhs` and `*this`.

27.5.2.3.2 Get area access [streambuf.get.area]

```
char_type* eback() const;
```

1 *Returns:* The beginning pointer for the input sequence.

```
char_type* gptr() const;
```

2 *Returns:* The next pointer for the input sequence.

```
char_type* egptr() const;
```

3 *Returns:* The end pointer for the input sequence.

```
void gbump(int n);
```

4 *Effects:* Adds `n` to the next pointer for the input sequence.

```
void setg(char_type* gbeg, char_type* gnext, char_type* gend);
```

5 *Postconditions:* `gbeg == eback()`, `gnext == gptr()`, and `gend == egptr()`.

27.5.2.3.3 Put area access [streambuf.put.area]

```
char_type* pbase() const;
```

1 *Returns:* The beginning pointer for the output sequence.

```
char_type* pptr() const;
```

2 *Returns:* The next pointer for the output sequence.

```
char_type* epptr() const;
```

3 *Returns:* The end pointer for the output sequence.

```
void pbump(int n);
```

4 *Effects:* Adds `n` to the next pointer for the output sequence.

```
void setp(char_type* pbeg, char_type* pend);
```

5 *Postconditions:* `pbeg == pbase()`, `pbeg == pptr()`, and `pend == epptr()`.

27.5.2.4 `basic_streambuf` virtual functions [streambuf.virtuals]

27.5.2.4.1 Locales [streambuf.virt.locales]

```
void imbue(const locale&)
```

1 *Effects:* Change any translations based on locale.

2 *Remarks:* Allows the derived class to be informed of changes in locale at the time they occur. Between invocations of this function a class derived from streambuf can safely cache results of calls to locale functions and to members of facets so obtained.

3 *Default behavior:* Does nothing.

27.5.2.4.2 Buffer management and positioning [streambuf.virt.buffer]

```
basic_streambuf* setbuf(char_type* s, streamsize n);
```

1 *Effects:* Influences stream buffering in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (27.7.1.4, 27.8.1.5).

2 *Default behavior:* Does nothing. Returns `this`.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

3 *Effects:* Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (27.7.1.4, 27.8.1.5).

4 *Default behavior:* Returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
                 ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

Effects: Alters the stream positions within one or more of the controlled sequences in a way that is defined separately for each class derived from `basic_streambuf` in this Clause (27.7.1, 27.8.1.1).

Default behavior: Returns `pos_type(off_type(-1))`.

```
int sync();
```

Effects: Synchronizes the controlled sequences with the arrays. That is, if `pbase()` is non-null the characters between `pbase()` and `pptr()` are written to the controlled sequence. The pointers may then be reset as appropriate.

Returns:-1 on failure. What constitutes failure is determined by each derived class (27.8.1.5).

Default behavior: Returns zero.

27.5.2.4.3 Get area [streambuf.virt.get]

```
streamsize showmanyc();
```[301]

Returns: an estimate of the number of characters available in the sequence, or -1. If it returns a positive value, then successive calls to `underflow()` will not return `traits::eof()` until at least that number of characters have been extracted from the stream. If `showmanyc()` returns -1, then calls to `underflow()` or `uflow()` will fail.[302]

Default behavior: Returns zero.

Remarks: Uses `traits::eof()`.

```
streamsize xsgetn(char_type* s, streamsize n);
```

Effects: Assigns up to `n` characters to successive elements of the array whose first element is designated by `s`. The characters assigned are read from the input sequence as if by repeated calls to `sbumpc()`. Assigning stops when either `n` characters have been assigned or a call to `sbumpc()` would return `traits::eof()`.

Returns: The number of characters assigned.[303]

Remarks: Uses `traits::eof()`.

```
int_type underflow();
```

Remarks: The public members of `basic_streambuf` call this virtual function only if `gptr()` is null or `gptr() >= egptr()`

Returns: `traits::to_int_type(c)`, where `c` is the first *character* of the *pending sequence*, without moving the input sequence position past it. If the pending sequence is null then the function returns `traits::eof()` to indicate failure.

The *pending sequence* of characters is defined as the concatenation of:

a) If `gptr()` is non- `NULL`, then the `egptr() - gptr()` characters starting at `gptr()`, otherwise the empty sequence.

b) Some sequence (possibly empty) of characters read from the input sequence.

301) `The morphemes of showmanycare ‘‘es-how-many-see’’, not ‘‘show-manic’’.`
302) `underflow` or `uflow` might fail by throwing an exception prematurely. The intention is not only that the calls will not return `eof()` but that they will return “immediately.”
303) Classes derived from `basic_streambuf` can provide more efficient ways to implement `xsgetn()` and `xsputn()` by overriding these definitions from the base class.

10 The *result character* is

a) If the pending sequence is non-empty, the first character of the sequence.

b) If the pending sequence is empty then the next character that would be read from the input sequence.

11 The *backup sequence* is defined as the concatenation of:

a) If `eback()` is null then empty,

b) Otherwise the `gptr() - eback()` characters beginning at `eback()`.

12 *Effects:* The function sets up the `gptr()` and `egptr()` satisfying one of:

a) If the pending sequence is non-empty, `egptr()` is non-null and `egptr() - gptr()` characters starting at `gptr()` are the characters in the pending sequence

b) If the pending sequence is empty, either `gptr()` is null or `gptr()` and `egptr()` are set to the same non-`NULL` pointer.

13 If `eback()` and `gptr()` are non-null then the function is not constrained as to their contents, but the "usual backup condition" is that either:

a) If the backup sequence contains at least `gptr() - eback()` characters, then the `gptr() - eback()` characters starting at `eback()` agree with the last `gptr() - eback()` characters of the backup sequence.

b) Or the `n` characters starting at `gptr() - n` agree with the backup sequence (where `n` is the length of the backup sequence)

14 *Default behavior:* Returns `traits::eof()`.

```
int_type uflow();
```

15 *Requires:* The constraints are the same as for `underflow()`, except that the result character shall be transferred from the pending sequence to the backup sequence, and the pending sequence shall not be empty before the transfer.

16 *Default behavior:* Calls `underflow()`. If `underflow()` returns `traits::eof()`, returns `traits::eof()`. Otherwise, returns the value of `traits::to_int_type(*gptr())` and increment the value of the next pointer for the input sequence.

17 *Returns:* `traits::eof()` to indicate failure.

27.5.2.4.4 Putback [streambuf.virt.pback]

```
int_type pbackfail(int_type c = traits::eof());
```

1 *Remarks:* The public functions of `basic_streambuf` call this virtual function only when `gptr()` is null, `gptr() == eback()`, or `traits::eq(traits::to_char_type(c),gptr()[-1])` returns `false`. Other calls shall also satisfy that constraint.

The *pending sequence* is defined as for `underflow()`, with the modifications that

— If `traits::eq_int_type(c,traits::eof())` returns `true`, then the input sequence is backed up one character before the pending sequence is determined.

— If `traits::eq_int_type(c,traits::eof())` return false, then `c` is prepended. Whether the input sequence is backed up or modified in any other way is unspecified.

2 *Postcondition:* On return, the constraints of `gptr()`, `eback()`, and `pptr()` are the same as for `underflow()`.

3 *Returns:* `traits::eof()` to indicate failure. Failure may occur because the input sequence could not be backed up, or if for some other reason the pointers could not be set consistent with the constraints. `pbackfail()` is called only when put back has really failed.

4 Returns some value other than `traits::eof()` to indicate success.

5 *Default behavior:* Returns `traits::eof()`.

27.5.2.4.5 Put area [streambuf.virt.put]

```
streamsize xsputn(const char_type* s, streamsize n);
```

1 *Effects:* Writes up to `n` characters to the output sequence as if by repeated calls to `sputc(c)`. The characters written are obtained from successive elements of the array whose first element is designated by `s`. Writing stops when either `n` characters have been written or a call to `sputc(c)` would return `traits::eof()`.

2 *Returns:* The number of characters written.

```
int_type overflow(int_type c = traits::eof());
```

3 *Effects:* Consumes some initial subsequence of the characters of the *pending sequence*. The pending sequence is defined as the concatenation of

a) if `pbase()` is `NULL` then the empty sequence otherwise, `pptr() - pbase()` characters beginning at `pbase()`.

b) if `traits::eq_int_type(c,traits::eof())` returns `true`, then the empty sequence otherwise, the sequence consisting of `c`.

4 *Remarks:* The member functions `sputc()` and `sputn()` call this function in case that no room can be found in the put buffer enough to accomodate the argument character sequence.

5 *Requires:* Every overriding definition of this virtual function shall obey the following constraints:

1) The effect of consuming a character on the associated output sequence is specified[304]

2) Let `r` be the number of characters in the pending sequence not consumed. If `r` is non-zero then `pbase()` and `pptr()` shall be set so that: `pptr() - pbase() == r` and the `r` characters starting at `pbase()` are the associated output stream. In case `r` is zero (all characters of the pending sequence have been consumed) then either `pbase()` is set to `NULL`, or `pbase()` and `pptr()` are both set to the same `NULL` non-value.

3) The function may fail if either appending some character to the associated output stream fails or if it is unable to establish `pbase()` and `pptr()` according to the above rules.

6 *Returns:* `traits::eof()` or throws an exception if the function fails.

Otherwise, returns some value other than `traits::eof()` to indicate success.[305]

7 *Default behavior:* Returns `traits::eof()`.

304) That is, for each class derived from an instance of `basic_streambuf` in this Clause (27.7.1, 27.8.1.1), a specification of how consuming a character effects the associated output sequence is given. There is no requirement on a program-defined class.
305) Typically, `overflow` returns `c` to indicate success, except when `traits::eq_int_type(c,traits::eof())` returns `true`, in which case it returns `traits::not_eof(c)`.

27.6 Formatting and manipulators [iostream.format]

Header <istream> synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_istream;
  typedef basic_istream<char>     istream;
  typedef basic_istream<wchar_t> wistream;

  template <class charT, class traits = char_traits<charT> >
    class basic_iostream;
  typedef basic_iostream<char>    iostream;
  typedef basic_iostream<wchar_t> wiostream;

  template <class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

Header <ostream> synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char>     ostream;
  typedef basic_ostream<wchar_t> wostream;

  template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
  template <class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
  template <class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}
```

Header <iomanip> synopsis

```
namespace std {
  // types T1, T2, ... are unspecified implementation types
  T1 resetiosflags(ios_base::fmtflags mask);
  T2 setiosflags  (ios_base::fmtflags mask);
  T3 setbase(int base);
  template<charT> T4 setfill(charT c);
  T5 setprecision(int n);
  T6 setw(int n);
  template <class moneyT> T7 get_money(moneyT& mon, bool intl = false);
  template <class charT, class moneyT> T8 put_money(const moneyT& mon, bool intl = false);
  template <class charT> T9 get_time(struct tm* tmb, const charT* fmt);
  template <class charT> T10 put_time(const struct tm* tmb, const charT* fmt);
}
```

27.6.1 Input streams [input.streams]

1 The header <istream> defines two types and a function signature that control input from a stream buffer.

27.6.1.1 Class template basic_istream [istream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_istream : virtual public basic_ios<charT,traits> {
  public:
    // types (inherited from basic_ios (27.4.4)):
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.6.1.1.1 Constructor/destructor:
    explicit basic_istream(basic_streambuf<charT,traits>* sb);
    basic_istream(basic_istream&& rhs);
    virtual ~basic_istream();

    // 27.6.1.1.2 Assign/swap:
    basic_istream& operator=(basic_istream&& rhs);
    void swap(basic_istream&& rhs);


    // 27.6.1.1.3 Prefix/suffix:
    class sentry;

    // 27.6.1.2 Formatted input:
    basic_istream<charT,traits>& operator>>(
      basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&));
    basic_istream<charT,traits>& operator>>(
                        basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
    basic_istream<charT,traits>& operator>>(
      ios_base& (*pf)(ios_base&));

    basic_istream<charT,traits>& operator>>(bool& n);
    basic_istream<charT,traits>& operator>>(short& n);
    basic_istream<charT,traits>& operator>>(unsigned short& n);
    basic_istream<charT,traits>& operator>>(int& n);
    basic_istream<charT,traits>& operator>>(unsigned int& n);
    basic_istream<charT,traits>& operator>>(long& n);
    basic_istream<charT,traits>& operator>>(unsigned long& n);
    basic_istream<charT,traits>& operator>>(long long& n);
    basic_istream<charT,traits>& operator>>(unsigned long long& n);
    basic_istream<charT,traits>& operator>>(float& f);
    basic_istream<charT,traits>& operator>>(double& f);
    basic_istream<charT,traits>& operator>>(long double& f);

    basic_istream<charT,traits>& operator>>(void*& p);
    basic_istream<charT,traits>& operator>>(
      basic_streambuf<char_type,traits>* sb);

    // 27.6.1.3 Unformatted input:
    streamsize gcount() const;
    int_type get();
    basic_istream<charT,traits>& get(char_type& c);
    basic_istream<charT,traits>& get(char_type* s, streamsize n);
    basic_istream<charT,traits>& get(char_type* s, streamsize n,
```

```
                                    char_type delim);
  basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb);
  basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
                                   char_type delim);

  basic_istream<charT,traits>& getline(char_type* s, streamsize n);
  basic_istream<charT,traits>& getline(char_type* s, streamsize n,
                                       char_type delim);

  basic_istream<charT,traits>& ignore(
    streamsize n = 1, int_type delim = traits::eof());
  int_type                     peek();
  basic_istream<charT,traits>& read    (char_type* s, streamsize n);
  streamsize                   readsome(char_type* s, streamsize n);

  basic_istream<charT,traits>& putback(char_type c);
  basic_istream<charT,traits>& unget();
  int sync();

  pos_type tellg();
  basic_istream<charT,traits>& seekg(pos_type);
  basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);
};

// 27.6.1.2.3 character extraction templates:
template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&&,
                                          charT&);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&&,
                                         unsigned char&);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&&,
                                         signed char&);

template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&&,
                                          charT*);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&&,
                                         unsigned char*);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&&,
                                         signed char*);

// swap:
template <class charT, class traits>
  void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_istream<charT, traits>&& x, basic_istream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>&& y);
}
```

1 The class `basic_istream` defines a number of member function signatures that assist in reading and inter-

preting input from sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted input functions* (or *extractors*) and the *unformatted input functions.* Both groups of input functions are described as if they obtain (or *extract*) input *characters* by calling `rdbuf()->sbumpc()` or `rdbuf()->sgetc()`. They may use other public members of `istream`.

3 If `rdbuf()->sbumpc()` or `rdbuf()->sgetc()` returns `traits::eof()`, then the input function, except as explicitly noted otherwise, completes its actions and does `setstate(eofbit)`, which may throw `ios_base::failure` (27.4.4.3), before returning.

4 If one of these called functions throws an exception, then unless explicitly noted otherwise, the input function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the input function rethrows the exception without completing its actions, otherwise it does not throw anything and proceeds as if the called function had returned a failure indication.

27.6.1.1.1 `basic_istream` constructors [istream.cons]

```
explicit basic_istream(basic_streambuf<charT,traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_istream`, assigning initial values to the base class by calling `basic_ios::init(sb)` (27.4.4.1).

2 *Postcondition:* `gcount() == 0`

```
basic_istream(basic_istream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class, copying the `gcount()` from `rhs`, calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class, and setting the `gcount()` for `rhs` to 0.

```
virtual ~basic_istream();
```

4 *Effects:* Destroys an object of class `basic_istream`.

5 *Remarks:* Does not perform any operations of `rdbuf()`.

27.6.1.1.2 Class `basic_istream` assign and swap [istream.assign]

```
basic_istream& operator=(basic_istream&& rhs);
```

1 *Effects:* `swap(rhs);`.

2 *Returns:* `*this`.

```
void swap(basic_istream&& rhs);
```

3 *Effects:* Calls `basic_ios<charT, traits>::swap(rhs)`. Exchanges the values returned by `gcount()` and `rhs.gcount()`.

```
template <class charT, class traits>
  void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_istream<charT, traits>&& x, basic_istream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_istream<charT, traits>& x, basic_istream<charT, traits>&& y);
```

4 *Effects:* `x.swap(y)`.

27.6.1.1.3 Class `basic_istream::sentry` [istream::sentry]

```
namespace std {
  template <class charT,class traits = char_traits<charT> >
  class basic_istream<charT,traits>::sentry {
    typedef traits traits_type;
    // bool ok_;                                              exposition only
  public:
    explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false);
    ~sentry();
    explicit operator bool() const { return ok_; }
    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

1 The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

`explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false);`

2 *Effects:* If `is.good()` is `true`, prepares for formatted or unformatted input. First, if `is.tie()` is not a null pointer, the function calls `is.tie()->flush()` to synchronize the output sequence with any associated external C stream. Except that this call can be suppressed if the put area of `is.tie()` is empty. Further an implementation is allowed to defer the call to `flush` until a call of `is.rdbuf()->underflow()` occurs. If no such call occurs before the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.[306] If `noskipws` is zero and `is.flags() & ios_base::skipws` is nonzero, the function extracts and discards each character as long as the next available input character `c` is a whitespace character. If `is.rdbuf()->sbumpc()` or `is.rdbuf()->sgetc()` returns `traits::eof()`, the function calls `setstate(failbit | eofbit)` (which may throw `ios_base::failure`).

3 *Remarks:* The constructor `explicit sentry(basic_istream<charT,traits>& is, bool noskipws = false)` uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

4 To decide if the character `c` is a whitespace character, the constructor performs "as if" it executes the following code fragment:

```
const ctype<charT>& ctype = use_facet<ctype<charT> >(is.getloc());
if (ctype.is(ctype.space,c)!=0)
  // c is a whitespace character.
```

5 If, after any preparation is completed, `is.good()` is `true`, `ok_ != false` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3))[307]

6 [*Note:* This paragraph is intentionally empty. — *end note*]

`~sentry();`

7 *Effects:* None.

`explicit operator bool() const;`

8 *Effects:* Returns `ok_`.

306) This will be possible only in functions that are part of the library. The semantics of the constructor used in user code is as specified.

307) The sentry constructor and destructor can also perform additional implementation-dependent operations.

27.6.1.2 Formatted input functions [istream.formatted]

27.6.1.2.1 Common requirements [istream.formatted.reqmts]

1 Each formatted input function begins execution by constructing an object of class `sentry` with the `noskipws` (second) argument `false`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. If an exception is thrown during input then `ios::badbit` is turned on[308] in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. In any case, the formatted input function destroys the `sentry` object. If no exception has been thrown, it returns `*this`.

27.6.1.2.2 Arithmetic Extractors [istream.formatted.arithmetic]

```
operator>>(unsigned short& val);
operator>>(unsigned int& val);
operator>>(long& val);
operator>>(unsigned long& val);
operator>>(long long& val);
operator>>(unsigned long long& val);
operator>>(float& val);
operator>>(double& val);
operator>>(long double& val);
operator>>(bool& val);
operator>>(void*& val);
```

1 As in the case of the inserters, these extractors depend on the locale's `num_get<>` (22.2.2.1) object to perform parsing the input stream data. These extractors behave as formatted input functions (as described in 27.6.1.2.1). After a sentry object is constructed, the conversion occurs as if performed by the following code fragment:

```
typedef num_get< charT,istreambuf_iterator<charT,traits> > numget;
iostate err = 0;
use_facet< numget >(loc).get(*this, 0, *this, err, val);
setstate(err);
```

In the above fragment, `loc` stands for the private member of the `basic_ios` class. [*Note:* The first argument provides an object of the `istreambuf_iterator` class which is an iterator pointed to an input stream. It bypasses istreams and uses streambufs directly. — *end note*] Class `locale` relies on this type as its interface to `istream`, so that it does not need to depend directly on `istream`.

```
operator>>(short& val);
```

2 The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
typedef num_get<charT,istreambuf_iterator<charT,traits> > numget;
iostate err = 0;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if (err != 0)
  ;
else if (lval < numeric_limits<short>::min()
  || numeric_limits<short>::max() < lval))
    err = ios_base::failbit;
```

308) This is done without causing an `ios::failure` to be thrown.

```
  else
    val = static_cast<short>(lval);
  setstate(err);

operator>>(int& val);
```

3 The conversion occurs as if performed by the following code fragment (using the same notation as for the preceding code fragment):

```
typedef num_get<charT,istreambuf_iterator<charT,traits> > numget;
iostate err = 0;
long lval;
use_facet<numget>(loc).get(*this, 0, *this, err, lval);
if err != 0)
  ;
else if (lval < numeric_limits<int>::min()
  || numeric_limits<int>::max() < lval))
    err = ios_base::failbit;
else
  val = static_cast<int>(lval);
setstate(err);
```

27.6.1.2.3 `basic_istream::operator>>` [istream::extractors]

```
basic_istream<charT,traits>& operator>>
    (basic_istream<charT,traits>& (*pf)(basic_istream<charT,traits>&))
```

1 *Effects:* None. This extractor does not behave as a formatted input function (as described in 27.6.1.2.1.)

2 *Returns:* `pf(*this)`.[309]

```
basic_istream<charT,traits>& operator>>
    (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
```

3 *Effects:* Calls `pf(*this)`. This extractor does not behave as a formatted input function (as described in 27.6.1.2.1).

4 *Returns:* `*this`.

```
basic_istream<charT,traits>& operator>>
    (ios_base& (*pf)(ios_base&));
```

5 *Effects:* Calls `pf(*this)`.[310] This extractor does not behave as a formatted input function (as described in 27.6.1.2.1).

6 *Returns:* `*this`.

```
template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&& in,
                                          charT* s);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&& in,
                                         unsigned char* s);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&& in,
```

309) See, for example, the function signature `ws(basic_istream&)` (27.6.1.4).
310) See, for example, the function signature `dec(ios_base&)` (27.4.5.3).

```
                                        signed char* s);
```

7 *Effects:* Behaves like a formatted input member (as described in 27.6.1.2.1) of `in`. After a `sentry` object is constructed, `operator>>` extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `width()` is greater than zero, `n` is `width()`. Otherwise `n` is the the number of elements of the largest array of `char_type` that can store a terminating `charT()`. `n` is the maximum number of characters stored.

8 Characters are extracted and stored until any of the following occurs:

— `n-1` characters are stored;

— end of file occurs on the input sequence;

— `ct.is(ct.space,c)` is `true` for the next available input character `c`, where `ct` is `use_facet<ctype< charT> >(in.getloc())`.

9 `operator>>` then stores a null byte (`charT()`) in the next position, which may be the first position if no characters were extracted. `operator>>` then calls `width(0)`.

10 If the function extracted no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3).

11 *Returns:* `in`.

```
template<class charT, class traits>
  basic_istream<charT,traits>& operator>>(basic_istream<charT,traits>&& in,
                                          charT& c);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&& in,
                                         unsigned char& c);
template<class traits>
  basic_istream<char,traits>& operator>>(basic_istream<char,traits>&& in,
                                         signed char& c);
```

12 *Effects:* Behaves like a formatted input member (as described in 27.6.1.2.1) of `in`. After a `sentry` object is constructed a character is extracted from `in`, if one is available, and stored in `c`. Otherwise, the function calls `in.setstate(failbit)`.

13 *Returns:* `in`.

```
basic_istream<charT,traits>& operator>>
  (basic_streambuf<charT,traits>* sb);
```

14 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). If `sb` is null, calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3). After a sentry object is constructed, extracts characters from `*this` and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted);

— an exception occurs (in which case the exception is caught).

15 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3). If it inserted no characters because it caught an exception thrown while extracting characters from `*this` and `failbit` is on in `exceptions()` (27.4.4.3), then the caught exception is rethrown.

16 *Returns:* `*this`.

27.6.1.3 Unformatted input functions [istream.unformatted]

1 Each unformatted input function begins execution by constructing an object of class `sentry` with the default argument `noskipws` (second) argument `true`. If the `sentry` object returns `true`, when converted to a value of type `bool`, the function endeavors to obtain the requested input. Otherwise, if the sentry constructor exits by throwing an exception or if the sentry object returns false, when converted to a value of type `bool`, the function returns without attempting to obtain any input. In either case the number of extracted characters is set to 0; unformatted input functions taking a character array of non-zero size as an argument shall also store a null character (using `charT()`) in the first location of the array. If an exception is thrown during input then `ios::badbit` is turned on[311] in `*this`'s error state. (Exceptions thrown from `basic_ios<>::clear()` are not caught or rethrown.) If `(exceptions()&badbit) != 0` then the exception is rethrown. It also counts the number of characters extracted. If no exception has been thrown it ends by storing the count in a member object and returning the value specified. In any event the `sentry` object is destroyed before leaving the unformatted input function.

```
streamsize gcount() const;
```

2 *Effects:* None. This member function does not behave as an unformatted input function (as described in 27.6.1.3, paragraph 1).

3 *Returns:* The number of characters extracted by the last unformatted input member function called for the object.

```
int_type get();
```

4 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts a character `c`, if one is available. Otherwise, the function calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3),

5 *Returns:* `c` if available, otherwise `traits::eof()`.

```
basic_istream<charT,traits>& get(char_type& c);
```

6 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts a character, if one is available, and assigns it to `c`.[312] Otherwise, the function calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

7 *Returns:* `*this`.

```
basic_istream<charT,traits>& get(char_type* s, streamsize n,
                 char_type delim );
```

8 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[313] Characters are extracted and stored until any of the following occurs:

— `n` is less than one or `n - 1` characters are stored;

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);

— `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted).

311) This is done without causing an `ios::failure` to be thrown.
312) Note that this function is not overloaded on types `signed char` and `unsigned char`.
313) Note that this function is not overloaded on types `signed char` and `unsigned char`.

9 If the function stores no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)). In any case, if `n` is greater than zero it then stores a null character into the next successive location of the array.

10 *Returns:* `*this`.

```
basic_istream<charT,traits>& get(char_type* s, streamsize n)
```

11 *Effects:* Calls `get(s,n,widen('\n'))`

12 *Returns:* Value returned by the call.

```
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb,
                  char_type delim );
```

13 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts characters and inserts them in the output sequence controlled by `sb`. Characters are extracted and inserted until any of the following occurs:

— end-of-file occurs on the input sequence;

— inserting in the output sequence fails (in which case the character to be inserted is not extracted);

— `traits::eq(c, delim)` for the next available input character `c` (in which case `c` is not extracted);

— an exception occurs (in which case, the exception is caught but not rethrown).

14 If the function inserts no characters, it calls `setstate(failbit)`, which may throw `ios_base::failure` (27.4.4.3).

15 *Returns:* `*this`.

```
basic_istream<charT,traits>& get(basic_streambuf<char_type,traits>& sb);
```

16 *Effects:* Calls `get(sb, widen('\n'))`

17 *Returns:* Value returned by the call.

```
basic_istream<charT,traits>& getline(char_type* s, streamsize n,
                    char_type delim);
```

18 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[314] Characters are extracted and stored until one of the following occurs:

1. end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`);
2. `traits::eq(c, delim)` for the next available input character `c` (in which case the input character is extracted but not stored);[315]
3. `n` is less than one or `n - 1` characters are stored (in which case the function calls `setstate(failbit)`).

19 These conditions are tested in the order shown.[316]

20 If the function extracts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).[317]

314) Note that this function is not overloaded on types `signed char` and `unsigned char`.
315) Since the final input character is "extracted," it is counted in the `gcount()`, even though it is not stored.
316) This allows an input line which exactly fills the buffer, without setting `failbit`. This is different behavior than the historical AT&T implementation.
317) This implies an empty input line will not cause `failbit` to be set.

21 In any case, if `n` is greater than zero, it then stores a null character (using `charT()`) into the next successive location of the array.

22 *Returns:* `*this`.

23 [*Example:*

```
#include <iostream>

int main() {
  using namespace std;
  const int line_buffer_size = 100;

  char buffer[line_buffer_size];
  int line_number = 0;
  while (cin.getline(buffer, line_buffer_size, '\n') || cin.gcount()) {
    int count = cin.gcount();
    if (cin.eof())
      cout << "Partial final line";   // cin.fail() is false
    else if (cin.fail()) {
      cout << "Partial long line";
      cin.clear(cin.rdstate() & ~ios_base::failbit);
    } else {
      count--;                         // Don't include newline in count
      cout << "Line " << ++line_number;
    }
    cout << " (" << count << " chars): " << buffer << endl;
  }
}
```

— *end example*]

```
basic_istream<charT,traits>& getline(char_type* s, streamsize n);
```

24 *Returns:* `getline(s,n,widen('\n'))`

```
basic_istream<charT,traits>&
    ignore(streamsize n = 1, int_type delim = traits::eof());
```

25 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, extracts characters and discards them. Characters are extracted until any of the following occurs:

— if `n != numeric_limits<streamsize>::max()` (18.2.1), `n` characters are extracted

— end-of-file occurs on the input sequence (in which case the function calls `setstate(eofbit)`, which may throw `ios_base::failure` (27.4.4.3));

— `traits::eq_int_type(traits::to_int_type(c), delim)` for the next available input character `c` (in which case `c` is extracted).

26 *Remarks:* The last condition will never occur if `traits::eq_int_type(delim, traits::eof())`.

27 *Returns:* `*this`.

```
int_type peek();
```

28 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, reads but does not extract the current input character.

29 *Returns:* `traits::eof()` if `good()` is `false`. Otherwise, returns `rdbuf()->sgetc()`.

```
basic_istream<charT,traits>& read(char_type* s, streamsize n);
```

30 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`.[318] Characters are extracted and stored until either of the following occurs:

— `n` characters are stored;

— end-of-file occurs on the input sequence (in which case the function calls `setstate(failbit|eofbit)`, which may throw `ios_base::failure` (27.4.4.3)).

31 *Returns:* `*this`.

```
streamsize readsome(char_type* s, streamsize n);
```

32 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. Otherwise extracts characters and stores them into successive locations of an array whose first element is designated by `s`. If `rdbuf()->in_avail() == -1`, calls `setstate(eofbit)` (which may throw `ios_base::failure` (27.4.4.3)), and extracts no characters;

— If `rdbuf()->in_avail() == 0`, extracts no characters

— If `rdbuf()->in_avail() > 0`, extracts `min(rdbuf()->in_avail(),n))`.

33 *Returns:* The number of characters extracted.

```
basic_istream<charT,traits>& putback(char_type c);
```

34 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf->sputbackc()`. If `rdbuf()` is null, or if `sputbackc()` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)). [*Note:* this function extracts no characters, so the value returned by the next call to `gcount()` is 0. — *end note*]

35 *Returns:* `*this`.

```
basic_istream<charT,traits>& unget();
```

36 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1). After constructing a sentry object, if `!good()` calls `setstate(failbit)` which may throw an exception, and return. If `rdbuf()` is not null, calls `rdbuf()->sungetc()`. If `rdbuf()` is null, or if `sungetc()` returns `traits::eof()`, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)). [*Note:* this function extracts no characters, so the value returned by the next call to `gcount()` is 0. — *end note*]

37 *Returns:* `*this`.

```
int sync();
```

38 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `rdbuf()` is a null pointer, returns

318) Note that this function is not overloaded on types `signed char` and `unsigned char`.

-1 . Otherwise, calls `rdbuf()->pubsync()` and, if that function returns -1 calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3), and returns `-1`. Otherwise, returns zero.

```
pos_type tellg();
```

39 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`.

40 *Returns:* After constructing a sentry object, if `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, in)`.

```
basic_istream<charT,traits>& seekg(pos_type pos);
```

41 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::in)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_basefailure`).

42 *Returns:* `*this`.

```
basic_istream<charT,traits>& seekg(off_type off, ios_base::seekdir dir);
```

43 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to `gcount()`. After constructing a sentry object, if `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::in)`.

44 *Returns:* `*this`.

27.6.1.4 Standard `basic_istream` manipulators [istream.manip]

```
namespace std {
  template <class charT, class traits>
    basic_istream<charT,traits>& ws(basic_istream<charT,traits>& is);
}
```

1 *Effects:* Behaves as an unformatted input function (as described in 27.6.1.3, paragraph 1), except that it does not count the number of characters extracted and does not affect the value returned by subsequent calls to is.gcount(). After constructing a sentry object extracts characters as long as the next available character `c` is whitespace or until there are no more characters in the sequence. Whitespace characters are distinguished with the same criterion as used by `sentry::sentry` (27.6.1.1.3). If `ws` stops extracting characters because there are no more available it sets `eofbit`, but not `failbit`.

2 *Returns:* `is`.

27.6.1.5 Class template `basic_iostream` [iostreamclass]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_iostream :
    public basic_istream<charT,traits>,
    public basic_ostream<charT,traits> {
  public:
    // types:
```

```
    typedef charT                    char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                   traits_type;

    // constructor/destructor
    explicit basic_iostream(basic_streambuf<charT,traits>* sb);
    basic_iostream(basic_iostream&& rhs);
    virtual ~basic_iostream();

    // assign/swap
    basic_iostream& operator=(basic_iostream&& rhs);
    void swap(basic_iostream&& rhs);
  };

  template <class charT, class traits>
    void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>& y);
  template <class charT, class traits>
    void swap(basic_iostream<charT, traits>&& x, basic_iostream<charT, traits>& y);
  template <class charT, class traits>
    void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>&& y);
}
```

1 The class `basic_iostream` inherits a number of functions that allow reading input and writing output to sequences controlled by a stream buffer.

27.6.1.5.1 `basic_iostream` constructors [iostream.cons]

```
explicit basic_iostream(basic_streambuf<charT,traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_iostream`, assigning initial values to the base classes by calling `basic_istream<charT,traits>(sb)` (27.6.1.1) and `basic_ostream<charT,traits>(sb)` (27.6.2.1)

2 *Postcondition:* `rdbuf()==sb` and `gcount()==0`.

```
basic_iostream(basic_iostream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs` by constructing the `basic_istream` base class with `move(rhs)`.

27.6.1.5.2 `basic_iostream` destructor [iostream.dest]

```
virtual ~basic_iostream();
```

1 *Effects:* Destroys an object of class `basic_iostream`.

2 *Remarks:* Does not perform any operations on `rdbuf()`.

27.6.1.5.3 `basic_iostream` assign and swap [iostream.assign]

```
basic_iostream& operator=(basic_iostream&& rhs);
```

1 *Effects:* `swap(rhs)`.

```
void swap(basic_iostream&& rhs);
```

2 *Effects:* Calls `basic_istream<charT, traits>::swap(rhs)`.

```
template <class charT, class traits>
  void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_iostream<charT, traits>&& x, basic_iostream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_iostream<charT, traits>& x, basic_iostream<charT, traits>&& y);
```

3 *Effects:* `x.swap(y)`.

27.6.2 Output streams [output.streams]

1 The header `<ostream>` defines a type and several function signatures that control output to a stream buffer.

27.6.2.1 Class template `basic_ostream` [ostream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ostream : virtual public basic_ios<charT,traits> {
  public:
    // types (inherited from basic_ios (27.4.4)):
    typedef charT                    char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                   traits_type;

    // 27.6.2.2 Constructor/destructor:
    explicit basic_ostream(basic_streambuf<char_type,traits>* sb);
    basic_ostream(basic_ostream&& rhs);
    virtual ~basic_ostream();

    // 27.6.2.3 Assign/swap
    basic_ostream& operator=(basic_ostream&& rhs);
    void swap(basic_ostream&& rhs);

    // 27.6.2.4 Prefix/suffix:
    class sentry;

    // 27.6.2.6 Formatted output:
    basic_ostream<charT,traits>& operator<<(
      basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&));
    basic_ostream<charT,traits>& operator<<(
      basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&));
    basic_ostream<charT,traits>& operator<<(
      ios_base& (*pf)(ios_base&));

    basic_ostream<charT,traits>& operator<<(bool n);
    basic_ostream<charT,traits>& operator<<(short n);
    basic_ostream<charT,traits>& operator<<(unsigned short n);
    basic_ostream<charT,traits>& operator<<(int n);
    basic_ostream<charT,traits>& operator<<(unsigned int n);
    basic_ostream<charT,traits>& operator<<(long n);
    basic_ostream<charT,traits>& operator<<(unsigned long n);
    basic_ostream<charT,traits>& operator<<(long long n);
```

```
  basic_ostream<charT,traits>& operator<<(unsigned long long n);
  basic_ostream<charT,traits>& operator<<(float f);
  basic_ostream<charT,traits>& operator<<(double f);
  basic_ostream<charT,traits>& operator<<(long double f);

  basic_ostream<charT,traits>& operator<<(const void* p);
  basic_ostream<charT,traits>& operator<<(
    basic_streambuf<char_type,traits>* sb);

  // 27.6.2.7 Unformatted output:
  basic_ostream<charT,traits>& put(char_type c);
  basic_ostream<charT,traits>& write(const char_type* s, streamsize n);

  basic_ostream<charT,traits>& flush();

  // 27.6.2.5 seeks:
  pos_type tellp();
  basic_ostream<charT,traits>& seekp(pos_type);
  basic_ostream<charT,traits>& seekp(off_type, ios_base::seekdir);
};

// 27.6.2.6.4 character inserters
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                          charT);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&&,
                                          charT);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                          char);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&&,
                                          char);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                         char);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&&,
                                         char);

// signed and unsigned
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                         signed char);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&&,
                                         signed char);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                         unsigned char)
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&&,
                                         unsigned char)
```

```
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                            const charT*);
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&&,
                                            const charT*);
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&,
                                            const char*);
  template<class charT, class traits>
    basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&&,
                                            const char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           const char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&&,
                                           const char*);
  // signed and unsigned
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           const signed char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&&,
                                           const signed char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&,
                                           const unsigned char*);
  template<class traits>
    basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&&,
                                           const unsigned char*);

  // swap:
  template <class charT, class traits>
    void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>& y);
  template <class charT, class traits>
    void swap(basic_ostream<charT, traits>&& x, basic_ostream<charT, traits>& y);
  template <class charT, class traits>
    void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>&& y);
}
```

1 The class `basic_ostream` defines a number of member function signatures that assist in formatting and writing output to output sequences controlled by a stream buffer.

2 Two groups of member function signatures share common properties: the *formatted output functions* (or *inserters*) and the *unformatted output functions*. Both groups of output functions generate (or *insert*) output *characters* by actions equivalent to calling `rdbuf()->sputc(int_type)`. They may use other public members of `basic_ostream` except that they shall not invoke any virtual members of `rdbuf()` except `overflow()`, `xsputn()`, and `sync()`.

3 If one of these called functions throws an exception, then unless explicitly noted otherwise the output function sets `badbit` in error state. If `badbit` is on in `exceptions()`, the output function rethrows the exception without completing its actions, otherwise it does not throw anything and treat as an error.

27.6.2.2 `basic_ostream` constructors [ostream.cons]

```
explicit basic_ostream(basic_streambuf<charT,traits>* sb);
```

1 *Effects:* Constructs an object of class `basic_ostream`, assigning initial values to the base class by calling `basic_ios<charT,traits>::init(sb)` (27.4.4.1).

2 *Postcondition:* `rdbuf() == sb`.

```
virtual ~basic_ostream();
```

3 *Effects:* Destroys an object of class `basic_ostream`.

4 *Remarks:* Does not perform any operations on `rdbuf()`.

```
basic_ostream(basic_ostream&& rhs);
```

5 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by default constructing the base class and calling `basic_ios<charT, traits>::move(rhs)` to initialize the base class.

27.6.2.3 Class `basic_ostream` assign and swap [ostream.assign]

```
basic_ostream& operator=(basic_ostream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_ostream&& rhs);
```

3 *Effects:* Calls `basic_ios<charT, traits>::swap(rhs)`.

```
template <class charT, class traits>
  void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_ostream<charT, traits>&& x, basic_ostream<charT, traits>& y);
template <class charT, class traits>
  void swap(basic_ostream<charT, traits>& x, basic_ostream<charT, traits>&& y);
```

4 *Effects:* `x.swap(y)`.

27.6.2.4 Class `basic_ostream::sentry` [ostream::sentry]

```
namespace std {
  template <class charT,class traits = char_traits<charT> >
  class basic_ostream<charT,traits>::sentry {
    // bool ok_;                                          exposition only
  public:
    explicit sentry(basic_ostream<charT,traits>& os);
    ~sentry();
    explicit operator bool() const { return ok_; }

    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

1 The class `sentry` defines a class that is responsible for doing exception safe prefix and suffix operations.

```
explicit sentry(basic_ostream<charT,traits>& os);
```

2 If `os.good()` is nonzero, prepares for formatted or unformatted output. If `os.tie()` is not a null pointer, calls `os.tie()->flush()`.[319]

3 If, after any preparation is completed, `os.good()` is `true`, `ok_ == true` otherwise, `ok_ == false`. During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3))[320]

```
~sentry();
```

4 If `((os.flags() & ios_base::unitbuf) && !uncaught_exception())` is `true`, calls `os.flush()`.

```
explicit operator bool() const;
```

5 *Effects:* Returns `ok_`.

27.6.2.5 `basic_ostream` seek members [ostream.seeks]

```
pos_type tellp();
```

1 *Returns:* if `fail() != false`, returns `pos_type(-1)` to indicate failure. Otherwise, returns `rdbuf()->pubseekoff(0, cur, out)`.

```
basic_ostream<charT,traits>& seekp(pos_type pos);
```

2 *Effects:* If `fail() != true`, executes `rdbuf()->pubseekpos(pos, ios_base::out)`. In case of failure, the function calls `setstate(failbit)` (which may throw `ios_base::failure`).

3 *Returns:* `*this`.

```
basic_ostream<charT,traits>& seekp(off_type off, ios_base::seekdir dir);
```

4 *Effects:* If `fail() != true`, executes `rdbuf()->pubseekoff(off, dir, ios_base::out)`.

5 *Returns:* `*this`.

27.6.2.6 Formatted output functions [ostream.formatted]

27.6.2.6.1 Common requirements [ostream.formatted.reqmts]

1 Each formatted output function begins execution by constructing an object of class `sentry`. If this object returns `true` when converted to a value of type `bool`, the function endeavors to generate the requested output. If the generation fails, then the formatted output function does `setstate(ios_base::failbit)`, which might throw an exception. If an exception is thrown during output, then `ios::badbit` is turned on[321] in `*this`'s error state. If `(exceptions()&badbit) != 0` then the exception is rethrown. Whether or not an exception is thrown, the `sentry` object is destroyed before leaving the formatted output function. If no exception is thrown, the result of the formatted output function is `*this`.

2 The descriptions of the individual formatted output operations describe how they perform output and do not mention the `sentry` object.

27.6.2.6.2 Arithmetic Inserters [ostream.inserters.arithmetic]

```
operator<<(bool val);
operator<<(short val);
operator<<(unsigned short val);
```

319) The call `os.tie()->flush()` does not necessarily occur if the function can determine that no synchronization is necessary.
320) The `sentry` constructor and destructor can also perform additional implementation-dependent operations.
321) without causing an `ios::failure` to be thrown.

```
operator<<(int val);
operator<<(unsigned int val);
operator<<(long val);
operator<<(unsigned long val);
operator<<(long long val);
operator<<(unsigned long long val);
operator<<(float val);
operator<<(double val);
operator<<(long double val);
operator<<(const void* val);
```

1 *Effects:* The classes `num_get<>` and `num_put<>` handle locale-dependent numeric formatting and parsing. These inserter functions use the imbued `locale` value to perform numeric formatting. When `val` is of type `bool`, `long`, `unsigned long`, `long long`, `unsigned long long`, `double`, `long double`, or `const void*`, the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(), val).failed();
```

When `val` is of type `short` the formatting conversion occurs as if it performed the following code fragment:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
      ? static_cast<long>(static_cast<unsigned short>(val))
      : static_cast<long>(val)).failed();
```

When `val` is of type `int` the formatting conversion occurs as if it performed the following code fragment:

```
ios_base::fmtflags baseflags = ios_base::flags() & ios_base::basefield;
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
    baseflags == ios_base::oct || baseflags == ios_base::hex
      ? static_cast<long>(static_cast<unsigned int>(val))
      : static_cast<long>(val)).failed();
```

When `val` is of type `unsigned short` or `unsigned int` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
      static_cast<unsigned long>(val)).failed();
```

When `val` is of type `float` the formatting conversion occurs as if it performed the following code fragment:

```
bool failed = use_facet<
  num_put<charT,ostreambuf_iterator<charT,traits> >
    >(getloc()).put(*this, *this, fill(),
      static_cast<double>(val)).failed();
```

2 The first argument provides an object of the `ostreambuf_iterator<>` class which is an iterator for class `basic_ostream<>`. It bypasses `ostream`s and uses `streambuf`s directly. Class `locale` relies on these types as its interface to iostreams, since for flexibility it has been abstracted away from direct dependence on `ostream`. The second parameter is a reference to the base subobject of type `ios_base`. It provides formatting specifications such as field width, and a locale from which to obtain other facets. If `failed` is `true` then does `setstate(badbit)`, which may throw an exception, and returns.

3 *Returns:* `*this`.

27.6.2.6.3 `basic_ostream::operator<<` [ostream.inserters]

```
basic_ostream<charT,traits>& operator<<
    (basic_ostream<charT,traits>& (*pf)(basic_ostream<charT,traits>&))
```

1 *Effects:* None. Does not behave as a formatted output function (as described in 27.6.2.6.1).

2 *Returns:* `pf(*this)`.[322]

```
basic_ostream<charT,traits>& operator<<
    (basic_ios<charT,traits>& (*pf)(basic_ios<charT,traits>&))
```

3 *Effects:* Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 27.6.2.6.1).

4 *Returns:* `*this`.[323]

```
basic_ostream<charT,traits>& operator<<
    (ios_base& (*pf)(ios_base&))
```

5 *Effects:* Calls `pf(*this)`. This inserter does not behave as a formatted output function (as described in 27.6.2.6.1).

6 *Returns:* `*this`.

```
basic_ostream<charT,traits>& operator<<
    (basic_streambuf<charT,traits>* sb);
```

7 *Effects:* Behaves as an unformatted output function (as described in 27.6.2.7, paragraph 1). After the sentry object is constructed, if `sb` is null calls `setstate(badbit)` (which may throw `ios_base::failure`).

8 Gets characters from `sb` and inserts them in `*this`. Characters are read from `sb` and inserted until any of the following occurs:

- end-of-file occurs on the input sequence;
- inserting in the output sequence fails (in which case the character to be inserted is not extracted);
- an exception occurs while getting a character from `sb`.

9 If the function inserts no characters, it calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)). If an exception was thrown while extracting a character, the function sets `failbit` in error state, and if `failbit` is on in `exceptions()` the caught exception is rethrown.

10 *Returns:* `*this`.

322) See, for example, the function signature `endl(basic_ostream&)` (27.6.2.8) .
323) See, for example, the function signature `dec(ios_base&)` (27.4.5.3).

27.6.2.6.4 Character inserter function templates [ostream.inserters.character]

```
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          charT c);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&& out,
                                          charT c);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          char c);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&& out,
                                          char c);
  // specialization
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         char c);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
                                         char c);
  // signed and unsigned
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         signed char c);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
                                         signed char c);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         unsigned char c);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
                                         unsigned char c);
```

1 *Effects:* Behaves like a formatted inserter (as described in 27.6.2.6.1) of `out`. After a `sentry` object is constructed it inserts characters. In case `c` has type `char` and the character type of the stream is not `char`, then the character to be inserted is `out.widen(c)`; otherwise the character is `c`.Padding is determined as described in 22.2.2.2.2. `width(0)` is called. The insertion character and any required padding are inserted into `out`.

2 *Returns:* `out`.

```
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          const charT* s);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&& out,
                                          const charT* s);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>& out,
                                          const char* s);
template<class charT, class traits>
  basic_ostream<charT,traits>& operator<<(basic_ostream<charT,traits>&& out,
                                          const char* s);
template<class traits>
```

```
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         const char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
                                         const char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         const signed char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
                                         const signed char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>& out,
                                         const unsigned char* s);
template<class traits>
  basic_ostream<char,traits>& operator<<(basic_ostream<char,traits>&& out,
                                         const unsigned char* s);
```

3 *Requires:* `s` shall not be a null pointer.

4 *Effects:* Behaves like a formatted inserter (as described in 27.6.2.6.1) of `out`. After a `sentry` object is constructed it inserts `n` characters starting at `s`, where `n` is the number that would be computed as if by:

- `traits::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const charT*`, and also for the overload where the first argument is of type `basic_ostream<char, traits>&` and the second is of type `const char*`,
- `std::char_traits<char>::length(s)` for the overload where the first argument is of type `basic_ostream<charT, traits>&` and the second is of type `const char*`,
- `traits::length(reinterpret_cast<const char*>(s))` for the other two overloads.

5 Padding is determined as described in 22.2.2.2.2. The `n` characters starting at `s` are widened using `out.widen` (27.4.4.2). The widened characters and any required padding are inserted into `out`. Calls `width(0)`.

6 *Returns:* `out`.

27.6.2.7 Unformatted output functions [ostream.unformatted]

1 Each unformatted output function begins execution by constructing an object of class `sentry`. If this object returns `true`, while converting to a value of type `bool`, the function endeavors to generate the requested output. If an exception is thrown during output, then `ios::badbit` is turned on[324] in `*this`'s error state. If `(exceptions() & badbit) != 0` then the exception is rethrown. In any case, the unformatted output function ends by destroying the sentry object, then, if no exception was thrown, returning the value specified for the unformatted output function.

```
basic_ostream<charT,traits>& put(char_type c);
```

2 *Effects:* Behaves as an unformatted output function (as described in 27.6.2.7, paragraph 1). After constructing a sentry object, inserts the character `c`, if possible.[325]

3 Otherwise, calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)).

324) without causing an `ios::failure` to be thrown.
325) Note that this function is not overloaded on types `signed char` and `unsigned char`.

4 *Returns:* `*this`.

```
basic_ostream& write(const char_type* s, streamsize n);
```

5 *Effects:* Behaves as an unformatted output function (as described in 27.6.2.7, paragraph 1). After constructing a sentry object, obtains characters to insert from successive locations of an array whose first element is designated by `s`.[326] Characters are inserted until either of the following occurs:

— `n` characters are inserted;

— inserting in the output sequence fails (in which case the function calls `setstate(badbit)`, which may throw `ios_base::failure` (27.4.4.3)).

6 *Returns:* `*this`.

```
basic_ostream& flush();
```

7 *Effects:* Behaves as an unformatted output function (as described in 27.6.2.6.1, paragraph 1). If `rdbuf()` is not a null pointer, constructs a sentry object. If this object returns `true` when converted to a value of type `bool` the function calls `rdbuf()->pubsync()`. If that function returns -1 calls `setstate(badbit)` (which may throw `ios_base::failure` (27.4.4.3)). Otherwise, if the sentry object returns `false`, does nothing.

8 *Returns:* `*this`.

27.6.2.8 Standard `basic_ostream` manipulators [ostream.manip]

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& endl(basic_ostream<charT,traits>& os);
}
```

1 *Effects:* Calls `os.put(os.widen('\n'))`, then `os.flush()`.

2 *Returns:* `os`.

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& ends(basic_ostream<charT,traits>& os);
}
```

3 *Effects:* Inserts a null character into the output sequence: calls `os.put(charT())`.

4 *Returns:* `os`.

```
namespace std {
  template <class charT, class traits>
    basic_ostream<charT,traits>& flush(basic_ostream<charT,traits>& os);
}
```

5 *Effects:* Calls `os.flush()`.

6 *Returns:* `os`.

326) Note that this function is not overloaded on types `signed char` and `unsigned char`.

27.6.3 Standard manipulators [std.manip]

1 The header <iomanip> defines several functions that support extractors and inserters that alter information maintained by class ios_base and its derived classes.

unspecified resetiosflags(ios_base::fmtflags mask);

2 *Returns:* An object of unspecified type such that if out is an object of type basic_ostream<charT,traits> then the expression out <<resetiosflags(mask) behaves as if it called f(out, mask), or if in is an object of type basic_istream<charT, traits> then the expression in >>resetiosflags(mask) behaves as if it called f(in, mask), where the function f is defined as:[327]

```
void f(ios_base& str, ios_base::fmtflags mask) {
  // reset specified flags
  str.setf(ios_base::fmtflags(0), mask);
}
```

The expression out <<resetiosflags(mask) shall have type basic_ostream<charT,traits>& and value out. The expression in >>resetiosflags(mask) shall have type basic_istream<charT,traits>& and value in.

unspecified setiosflags(ios_base::fmtflags mask);

3 *Returns:* An object of unspecified type such that if out is an object of type basic_ostream<charT,traits> then the expression out <<setiosflags(mask) behaves as if it called f(out, mask), or if in is an object of type basic_istream<charT, traits> then the expression in >>setiosflags(mask) behaves as if it called f(in, mask), where the function f is defined as:

```
void f(ios_base& str, ios_base::fmtflags mask) {
  // set specified flags
  str.setf(mask);
}
```

The expression out <<setiosflags(mask) shall have type basic_ostream<charT, traits>& and value out. The expression in >>setiosflags(mask) shall have type basic_istream<charT, traits>& and value in.

unspecified setbase(int base);

4 *Returns:* An object of unspecified type such that if out is an object of type basic_ostream<charT, traits> then the expression out <<setbase(base) behaves as if it called f(out, base), or if in is an object of type basic_istream<charT, traits> then the expression in >>setbase(base) behaves as if it called f(in, base), where the function f is defined as:

```
void f(ios_base& str, int base) {
  // set basefield
  str.setf(base ==  8 ? ios_base::oct :
      base == 10 ? ios_base::dec :
      base == 16 ? ios_base::hex :
      ios_base::fmtflags(0), ios_base::basefield);
}
```

327) The expression cin >>resetiosflags(ios_base::skipws) clears ios_base::skipws in the format flags stored in the basic_istream<charT,traits> object cin (the same as cin >>noskipws), and the expression cout <<resetiosflags(ios_base::showbase) clears ios_base::showbase in the format flags stored in the basic_ostream<charT,traits> object cout (the same as cout <<noshowbase).

The expression `out <<setbase(base)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >>setbase(base)` shall have type `basic_istream<charT, traits>&` and value `in`.

unspecified `setfill(char_type c);`

5 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT,traits>` and `c` has type `charT` then the expression `out <<setfill(c)` behaves as if it called `f(out, c)`, where the function `f` is defined as:

```
template<class charT, class traits>
void f(basic_ios<charT,traits>& str, charT c) {
  // set fill character
  str.fill(c);
}
```

The expression `out <<setfill(c)` shall have type `basic_ostream<charT, traits>&` and value `out`.

unspecified `setprecision(int n);`

6 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out <<setprecision(n)` behaves as if it called `f(out, n)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >>setprecision(n)` behaves as if it called `f(in, n)`, where the function `f` is defined as:

```
void f(ios_base& str, int n) {
  // set precision
  str.precision(n);
}
```

The expression `out <<setprecision(n)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >>setprecision(n)` shall have type `basic_istream<charT, traits>&` and value `in`.

unspecified `setw(int n);`

7 *Returns:* An object of unspecified type such that if `out` is an instance of `basic_ostream<charT, traits>` then the expression `out <<setw(n)` behaves as if it called `f(out, n)`, or if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >>setw(n)` behaves as if it called `f(in, n)` , where the function `f` is defined as:

```
void f(ios_base& str, int n) {
  // set width
  str.width(n);
}
```

The expression `out <<setw(n)` shall have type `basic_ostream<charT, traits>&` and value `out`. The expression `in >>setw(n)` shall have type `basic_istream<charT, traits>&` and value `in`.

27.6.4 Extended Manipulators [ext.manip]

1 The header `<iomanip>` defines several functions that support extractors and inserters that allow for the parsing and formatting of sequences and values for money and time.

`template <class moneyT>` *unspecified* `get_money(moneyT& mon, bool intl = false);`

2 *Requires:* The type `moneyT` shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

3 *Effects:* The expression `in >> get_money(mon, intl)` described below behaves as a formatted input function (27.6.1.2.1).

4 *Returns:* An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >>get_money(mon, intl)` behaves as if it called `f(in, mon, intl)`, where the function `f` is defined as:

```
template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, moneyT& mon, bool intl) {
  typedef istreambuf_iterator<charT> Iter;
  typedef money_get<charT, Iter> MoneyGet;

  ios_base::iostate err = ios_base::goodbit;
  const MoneyGet &mg = use_facet<MoneyGet>(str.getloc());

  mg.get(Iter(str.rdbuf()), Iter(), intl, str, err, mon);

  if (ios_base::goodbit != err)
    str.setstate(err);
}
```

The expression `in >>get_money(mon, intl)` shall have type `basic_istream<charT, traits>&` and value `in`.

```
template <class charT, class moneyT> unspecified  put_money(const moneyT& mon, bool intl = false);
```

5 *Requires:* The type `moneyT` shall be either `long double` or a specialization of the `basic_string` template (Clause 21).

6 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out <<put_money(mon, intl)` behaves as a formatted input function that calls `f(out, mon, intl)`, where the function `f` is defined as:

```
template <class charT, class traits, class moneyT>
void f(basic_ios<charT, traits>& str, const moneyT& mon, bool intl) {
  typedef ostreambuf_iterator<charT> Iter;
  typedef money_put<charT, Iter> MoneyPut;

  const MoneyPut &mp = use_facet<MoneyPut>(str.getloc());
  const Iter end = mp.put(Iter(str.rdbuf()), intl, str, str.fill(), mon);

  if (end.failed())
    str.setstate(ios::badbit);
}
```

The expression `out <<put_money(mon, intl)` shall have type `basic_ostream<charT, traits>&` and value `out`.

```
template <class charT> unspecified  get_time(struct tm* tmb, const charT* fmt);
```

7 *Requires:* The argument `tmb` shall be a valid pointer to an object of type `struct tm`, and the argument `fmt` shall be a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

8 *Returns:* An object of unspecified type such that if `in` is an object of type `basic_istream<charT, traits>` then the expression `in >>get_time(tmb, fmt)` behaves as if it called `f(in, tmb, fmt)`, where the function `f` is defined as:

```
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, struct tm* tmb, const charT* fmt) {
  typedef streambuf_iterator<charT> Iter;
  typedef time_get<charT, Iter> TimeGet;

  ios_base::iostate err = ios_base::goodbit;
  const TimeGet& tg = use_facet<TimeGet>(str.getloc());

  tm.get(Iter(str.rdbuf()), Iter(), str, err, tmb,
    fmt, fmt + traits::length(fmt));

  if (err != ios_base::goodbit)
    str.setstate(err):
}
```

The expression `in >>get_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value `in`.

```
template <class charT> unspecified  put_time(const struct tm* tmb, const charT* fmt);
```

9 *Requires:* The argument `tmb` shall be a valid pointer to an object of type `struct tm`, and the argument `fmt` shall be a valid pointer to an array of objects of type `charT` with `char_traits<charT>::length(fmt)` elements.

10 *Returns:* An object of unspecified type such that if `out` is an object of type `basic_ostream<charT, traits>` then the expression `out <<put_time(tmb, fmt)` behaves as if it called `f(out, tmb, fmt)`, where the function `f` is defined as:

```
template <class charT, class traits>
void f(basic_ios<charT, traits>& str, const struct tm* tmb, const charT* fmt) {
  typedef ostreambuf_iterator<charT> Iter;
  typedef time_put<charT, Iter> TimePut;

  const TimePut& tp = use_facet<TimePut>(str.getloc());
  const Iter end = tp.put(Iter(str.rdbuf()), str, str.fill(), tmb,
    fmt, fmt + traits::length(fmt));

  if (end.failed())
    str.setstate(ios_base::badbit);
}
```

The expression `out <<put_time(tmb, fmt)` shall have type `basic_istream<charT, traits>&` and value `out`.

27.7 String-based streams [string.streams]

1 The header `<sstream>` defines four class templates and eight types that associate stream buffers with objects of class `basic_string`, as described in 21.2.

Header <sstream> synopsis

```
namespace std {
```

```
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_stringbuf;

  typedef basic_stringbuf<char>     stringbuf;
  typedef basic_stringbuf<wchar_t> wstringbuf;

  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_istringstream;

  typedef basic_istringstream<char>     istringstream;
  typedef basic_istringstream<wchar_t> wistringstream;

  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_ostringstream;
  typedef basic_ostringstream<char>     ostringstream;
  typedef basic_ostringstream<wchar_t> wostringstream;

  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
    class basic_stringstream;
  typedef basic_stringstream<char>     stringstream;
  typedef basic_stringstream<wchar_t> wstringstream;
}
```

27.7.1 Class template `basic_stringbuf` [stringbuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
      class Allocator = allocator<charT> >
  class basic_stringbuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // 27.7.1.1 Constructors:
    explicit basic_stringbuf(ios_base::openmode which
                             = ios_base::in | ios_base::out);
    explicit basic_stringbuf
    (const basic_string<charT,traits,Allocator>& str,
     ios_base::openmode which = ios_base::in | ios_base::out);
    basic_stringbuf(basic_stringbuf&& rhs);

    // 27.7.1.2 Assign and swap:
    basic_stringbuf& operator=(basic_stringbuf&& rhs);
    void swap(basic_stringbuf&& rhs);

    // 27.7.1.3 Get and set:
    basic_string<charT,traits,Allocator> str() const;
```

```
    void str(const basic_string<charT,traits,Allocator>& s);

  protected:
    // 27.7.1.4 Overridden virtual functions:
    virtual int_type   underflow();
    virtual int_type   pbackfail(int_type c = traits::eof());
    virtual int_type   overflow (int_type c = traits::eof());
    virtual  basic_streambuf<charT,traits>* setbuf(charT*, streamsize);


    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);

  private:
    // ios_base::openmode mode;                                    exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_stringbuf<charT, traits, Allocator>& x,
            basic_stringbuf<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_stringbuf<charT, traits, Allocator>&& x,
            basic_stringbuf<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_stringbuf<charT, traits, Allocator>& x,
            basic_stringbuf<charT, traits, Allocator>&& y);
}
```

1 The class `basic_stringbuf` is derived from `basic_streambuf` to associate possibly the input sequence and possibly the output sequence with a sequence of arbitrary *characters*. The sequence can be initialized from, or made available as, an object of class `basic_string`.

2 For the sake of exposition, the maintained data is presented here as:

— `ios_base::openmode mode`, has `in` set if the input sequence can be read, and `out` set if the output sequence can be written.

27.7.1.1 `basic_stringbuf` constructors [stringbuf.cons]

```
explicit basic_stringbuf(ios_base::openmode which =
                         ios_base::in | ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (27.5.2.1), and initializing `mode` with `which`.

2 *Postcondition:* `str() == ""`.

```
explicit basic_stringbuf(const basic_string<charT,traits,Allocator>& s,
                         ios_base::openmode which = ios_base::in | ios_base::out);
```

3 *Effects:* Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` (27.5.2.1), and initializing `mode` with `which`. Then calls `str(s)`.

```
basic_stringbuf(basic_stringbuf&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. The openmode, locale and any other state of `rhs` is also copied.

5 *Postconditions:* Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.

- `str() == rhs_p.str()`
- `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`
- `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`
- `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`
- `epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`
- `if (eback()) eback() != rhs_a.eback()`
- `if (gptr()) gptr() != rhs_a.gptr()`
- `if (egptr()) egptr() != rhs_a.egptr()`
- `if (pbase()) pbase() != rhs_a.pbase()`
- `if (pptr()) pptr() != rhs_a.pptr()`
- `if (epptr()) epptr() != rhs_a.epptr()`

27.7.1.2 Assign and swap [stringbuf.assign]

```
basic_stringbuf& operator=(basic_stringbuf&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_stringbuf&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
          basic_stringbuf<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>&& x,
          basic_stringbuf<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringbuf<charT, traits, Allocator>& x,
          basic_stringbuf<charT, traits, Allocator>&& y);
```

4 *Effects:* `x.swap(y)`.

27.7.1.3 Member functions [stringbuf.members]

```
basic_string<charT,traits,Allocator> str() const;
```

1 *Returns:* A `basic_string` object whose content is equal to the `basic_stringbuf` underlying character sequence. If the `basic_stringbuf` was created only in input mode, the resultant `basic_string` contains the character sequence in the range `[eback(),egptr())`. If the `basic_stringbuf` was created with `which & ios_base::out` being true then the resultant `basic_string` contains the character sequence in the range `[pbase(),high_mark)`, where `high_mark` represents the position one past the highest initialized character in the buffer. Characters can be initialized by writing to the stream, by constructing the `basic_stringbuf` with a `basic_string`, or by calling the `str(basic_string)` member function. In the case of calling the `str(basic_string)` member function, all characters initialized prior to the call are now considered uninitialized (except for those characters re-initialized by the new `basic_string`). Otherwise the `basic_stringbuf` has been created in neither input nor output mode and a zero length `basic_string` is returned.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

2 *Effects:* Copies the content of `s` into the `basic_stringbuf` underlying character sequence and initializes the input and output sequences according to `mode`.

3 *Postconditions:* If `mode & ios_base::out` is true, `pbase()` points to the first underlying character and `epptr() >= pbase() + s.size()` holds; in addition, if `mode & ios_base::in` is true, `pptr() == pbase() + s.data()` holds, otherwise `pptr() == pbase()` is true. If `mode & ios_base::in` is true, `eback()` points to the first underlying character, and both `gptr() == eback()` and `egptr() == eback() + s.size()` hold.

27.7.1.4 Overridden virtual functions [stringbuf.virtuals]

```
int_type underflow();
```

1 *Returns:* If the input sequence has a read position available, returns `traits::to_int_type(*gptr())`. Otherwise, returns `traits::eof()`. Any character in the underlying buffer which has been initialized is considered to be part of the input sequence.

```
int_type pbackfail(int_type c = traits::eof());
```

2 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

— If `traits::eq_int_type(c,traits::eof())` returns `false` and if the input sequence has a putback position available, and if `traits::eq(to_char_type(c),gptr()[-1])` returns `true`, assigns `gptr() - 1` to `gptr()`.

Returns: `c`.

— If `traits::eq_int_type(c,traits::eof())` returns `false` and if the input sequence has a putback position available, and if `mode & ios_base::out` is nonzero, assigns `c` to `*--gptr()`.

Returns: `c`.

— If `traits::eq_int_type(c,traits::eof())` returns `true` and if the input sequence has a putback position available, assigns `gptr() - 1` to `gptr()`.

Returns: `traits::not_eof(c)`.

3 *Returns:* `traits::eof()` to indicate failure.

4 *Remarks:* If the function can succeed in more than one of these ways, it is unspecified which way is chosen.

```
int_type overflow(int_type c = traits::eof());
```

Effects: Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

— If `traits::eq_int_type(c,traits::eof())` returns `false` and if either the output sequence has a write position available or the function makes a write position available (as described below), the function calls `sputc(c )`.

Signals success by returning `c`.

— If `traits::eq_int_type(c,traits::eof())` returns `true`, there is no character to append.

Signals success by returning a value other than `traits::eof()`.

Remarks: The function can alter the number of write positions available as a result of any call.

Returns: `traits::eof()` to indicate failure.

The function can make a write position available only if `(mode & ios_base::out) != 0`. To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements to hold the current array object (if any), plus at least one additional write position. If `(mode & ios_base::in) != 0`, the function alters the read end pointer `egptr()` to point just past the new write position.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
                 ios_base::openmode which
                   = ios_base::in | ios_base::out);
```

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 107.

Table 107 — `seekoff` positioning

Conditions	Result
`(which & ios_base::in) == ios_base::in`	positions the input sequence
`(which & ios_base::out) == ios_base::out`	positions the output sequence
`(which & (ios_base::in | ios_base::out)) == (ios_base::in | ios_base::out))` and `way ==` either `ios_base::beg` or `ios_base::end`	positions both the input and the output sequences
Otherwise	the positioning operation fails.

For a sequence to be positioned, if its next pointer (either `gptr()` or `pptr()`) is a null pointer and the new offset `newoff` is nonzero, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 108.

If `(newoff + off) < 0`, or if `newoff + off` refers to an uninitialized character (as defined in 27.7.1.3 paragraph 1), the positioning operation fails. Otherwise, the function assigns `xbeg + newoff + off` to the next pointer `xnext`.

Returns: `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

Table 108 — `newoff` values

Condition	`newoff` Value
`way == ios_base::beg`	0
`way == ios_base::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios_base::end`	the high mark pointer minus the beginning pointer (`high_mark - xbeg`).

```
pos_type seekpos(pos_type sp, ios_base::openmode which
                  = ios_base::in | ios_base::out);
```

13 *Effects:* Alters the stream position within the controlled sequences, if possible, to correspond to the stream position stored in `sp` (as described below).

- If `(which & ios_base::in) != 0`, positions the input sequence.
- If `(which & ios_base::out) != 0`, positions the output sequence.
- If `sp` is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If `sp` has not been obtained by a previous successful call to one of the positioning functions (`seekoff`, `seekpos`, `tellg`, `tellp`) the effect is undefined.

14 *Returns:* `sp` to indicate success, or `pos_type(off_type(-1))` to indicate failure.

```
basic_streambuf<charT,traits>* setbuf(charT* s, streamsize n);
```

15 *Effects:* implementation-defined, except that `setbuf(0,0)` has no effect.

16 *Returns:* `this`.

27.7.2 Class template `basic_istringstream` [istringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
  class basic_istringstream : public basic_istream<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // 27.7.2.1 Constructors:
    explicit basic_istringstream(ios_base::openmode which = ios_base::in);
    explicit basic_istringstream(
               const basic_string<charT,traits,Allocator>& str,
               ios_base::openmode which = ios_base::in);
    basic_istringstream(basic_istringstream&& rhs);

    // 27.7.2.2 Assign and swap:
    basic_istringstream& operator=(basic_istringstream&& rhs);
    void swap(basic_istringstream&& rhs);
```

```
    // 27.7.2.3 Members:
    basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& s);
  private:
    // basic_stringbuf<charT,traits,Allocator> sb;                    exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_istringstream<charT, traits, Allocator>& x,
            basic_istringstream<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_istringstream<charT, traits, Allocator>&& x,
            basic_istringstream<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_istringstream<charT, traits, Allocator>& x,
            basic_istringstream<charT, traits, Allocator>&& y);
}
```

1 The class `basic_istringstream<charT, traits, Allocator>` supports reading objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `stringbuf` object.

27.7.2.1 `basic_istringstream` constructors [istringstream.cons]

```
explicit basic_istringstream(ios_base::openmode which = ios_base::in);
```

1 *Effects:* Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in))` (27.7.1.1).

```
explicit basic_istringstream(
               const basic_string<charT,traits,allocator>& str,
               ios_base::openmode which = ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in))` (27.7.1.1).

```
basic_istringstream(basic_istringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

27.7.2.2 Assign and swap [istringstream.assign]

```
basic_istringstream& operator=(basic_istringstream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_istringstream&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
          basic_istringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>&& x,
          basic_istringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_istringstream<charT, traits, Allocator>& x,
          basic_istringstream<charT, traits, Allocator>&& y);
```

4 *Effects:* `x.swap(y)`.

27.7.2.3 Member functions [istringstream.members]

```
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)`.

```
basic_string<charT,traits,Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

3 *Effects:* Calls `rdbuf()->str(s)`.

27.7.3 Class `basic_ostringstream` [ostringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
  class basic_ostringstream : public basic_ostream<charT,traits> {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // 27.7.3.1 Constructors/destructor:
    explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
    explicit basic_ostringstream(
             const basic_string<charT,traits,Allocator>& str,
             ios_base::openmode which = ios_base::out);
    basic_ostringstream(basic_ostringstream&& rhs);

    // 27.7.3.2 Assign/swap:
    basic_ostringstream& operator=(basic_ostringstream&& rhs);
    void swap(basic_ostringstream&& rhs);
```

```
    // 27.7.3.3 Members:
    basic_stringbuf<charT,traits,Allocator>* rdbuf() const;

    basic_string<charT,traits,Allocator> str() const;
    void    str(const basic_string<charT,traits,Allocator>& s);
  private:
    // basic_stringbuf<charT,traits,Allocator> sb;                    exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_ostringstream<charT, traits, Allocator>& x,
            basic_ostringstream<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_ostringstream<charT, traits, Allocator>&& x,
            basic_ostringstream<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_ostringstream<charT, traits, Allocator>& x,
            basic_ostringstream<charT, traits, Allocator>&& y);
}
```

1 The class `basic_ostringstream<charT, traits, Allocator>` supports writing objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf` object to control the associated storage. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `stringbuf` object.

27.7.3.1 `basic_ostringstream` constructors [ostringstream.cons]

```
explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
```

1 *Effects:* Constructs an object of class `basic_ostringstream`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::out))` (27.7.1.1).

```
explicit basic_ostringstream(
  const basic_string<charT,traits,Allocator>& str,
  ios_base::openmode which = ios_base::out);
```

2 *Effects:* Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out))` (27.7.1.1).

```
basic_ostringstream(basic_ostringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_ostream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

27.7.3.2 Assign and swap [ostringstream.assign]

```
basic_ostringstream& operator=(basic_ostringstream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_ostringstream&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
          basic_ostringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>&& x,
          basic_ostringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_ostringstream<charT, traits, Allocator>& x,
          basic_ostringstream<charT, traits, Allocator>&& y);
```

4 *Effects:* `x.swap(y)`.

27.7.3.3 Member functions [ostringstream.members]

```
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)`.

```
basic_string<charT,traits,Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

3 *Effects:* Calls `rdbuf()->str(s)`.

27.7.4 Class template `basic_stringstream` [stringstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT>,
        class Allocator = allocator<charT> >
  class basic_stringstream
    : public basic_iostream<charT,traits> {
  public:

    // types:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;
    typedef Allocator                 allocator_type;

    // constructors/destructor
    explicit basic_stringstream(
    ios_base::openmode which = ios_base::out|ios_base::in);
    explicit basic_stringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out|ios_base::in);
    basic_stringstream(basic_stringstream&& rhs);

    // 27.7.5.1 Assign/swap:
```

```
    basic_stringstream& operator=(basic_stringstream&& rhs);
    void swap(basic_stringstream&& rhs);

    // Members:
    basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
    basic_string<charT,traits,Allocator> str() const;
    void str(const basic_string<charT,traits,Allocator>& str);

  private:
    // basic_stringbuf<charT, traits> sb;                    exposition only
  };

  template <class charT, class traits, class Allocator>
  void swap(basic_stringstream<charT, traits, Allocator>& x,
            basic_stringstream<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_stringstream<charT, traits, Allocator>&& x,
            basic_stringstream<charT, traits, Allocator>& y);
  template <class charT, class traits, class Allocator>
  void swap(basic_stringstream<charT, traits, Allocator>& x,
            basic_stringstream<charT, traits, Allocator>&& y);
}
```

1 The class template `basic_stringstream<charT, traits>` supports reading and writing from objects of class `basic_string<charT, traits, Allocator>`. It uses a `basic_stringbuf<charT, traits, Allocator>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as

— `sb`, the `stringbuf` object.

27.7.5 basic_stringstream constructors [stringstream.cons]

```
explicit basic_stringstream(
    ios_base::openmode which = ios_base::out|ios_base::in);
```

1 *Effects:* Constructs an object of class `basic_stringstream<charT,traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT,traits,Allocator>(which)`.

```
explicit basic_stringstream(
    const basic_string<charT,traits,Allocator>& str,
    ios_base::openmode which = ios_base::out|ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

```
basic_stringstream(basic_stringstream&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_stringbuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_stringbuf`.

27.7.5.1 Assign and swap [stringstream.assign]

```
basic_stringstream& operator=(basic_stringstream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_stringstream&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>& x,
          basic_stringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>&& x,
          basic_stringstream<charT, traits, Allocator>& y);
template <class charT, class traits, class Allocator>
void swap(basic_stringstream<charT, traits, Allocator>& x,
          basic_stringstream<charT, traits, Allocator>&& y);
```

4 *Effects:* `x.swap(y)`.

27.7.6 Member functions [stringstream.members]

```
basic_stringbuf<charT,traits,Allocator>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_stringbuf<charT,traits,Allocator>*>(&sb)`

```
basic_string<charT,traits,Allocator> str() const;
```

2 *Returns:* `rdbuf()->str()`.

```
void str(const basic_string<charT,traits,Allocator>& str);
```

3 *Effects:* Calls `rdbuf()->str(str)`.

27.8 File-based streams [file.streams]

27.8.1 File streams [fstreams]

1 The header `<fstream>` defines four class templates and eight types that associate stream buffers with files and assist reading and writing files.

Header `<fstream>` synopsis

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
    class basic_filebuf;
  typedef basic_filebuf<char>    filebuf;
  typedef basic_filebuf<wchar_t> wfilebuf;

  template <class charT, class traits = char_traits<charT> >
    class basic_ifstream;
  typedef basic_ifstream<char>    ifstream;
  typedef basic_ifstream<wchar_t> wifstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_ofstream;
  typedef basic_ofstream<char>    ofstream;
```

```
  typedef basic_ofstream<wchar_t> wofstream;

  template <class charT, class traits = char_traits<charT> >
    class basic_fstream;
  typedef basic_fstream<char>     fstream;
  typedef basic_fstream<wchar_t> wfstream;
}
```

2 In this subclause, the type name `FILE` refers to the type `FILE` declared in `<cstdio>` (27.8.2).[328]

File A File provides an external source/sink stream whose *underlaid character type* is `char` (byte).[329]

Multibyte character and Files A File provides byte sequences. So the streambuf (or its derived classes) treats a file as the external source/sink byte sequence. In a large character set environment, multibyte character sequences are held in files. In order to provide the contents of a file as wide character sequences, wide-oriented filebuf, namely wfilebuf should convert wide character sequences.

27.8.1.1 Class template `basic_filebuf` [filebuf]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.8.1.2 Constructors/destructor:
    basic_filebuf();
    basic_filebuf(basic_filebuf&& rhs);
    virtual ~basic_filebuf();

    // 27.8.1.3 Assign/swap:
    basic_filebuf& operator=(basic_filebuf&& rhs);
    void swap(basic_filebuf&& rhs);

     // 27.8.1.4 Members:
    bool is_open() const;
    basic_filebuf<charT,traits>* open(const char* s,
        ios_base::openmode mode);
    basic_filebuf<charT,traits>* open(const string& s,
        ios_base::openmode mode);
    basic_filebuf<charT,traits>* close();

  protected:
    // 27.8.1.5 Overridden virtual functions:
    virtual streamsize showmanyc();
    virtual int_type underflow();
    virtual int_type uflow();
    virtual int_type pbackfail(int_type c = traits::eof());
```

328) In C `FILE` must be a `typedef`. In C++ it may be a `typedef` or other type name.
329) A File is a sequence of multibyte characters. In order to provide the contents as a wide character sequence, `filebuf` should convert between wide character sequences and multibyte character sequences.

```
    virtual int_type overflow (int_type c = traits::eof());

    virtual basic_streambuf<charT,traits>*
        setbuf(char_type* s, streamsize n);
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp,
        ios_base::openmode which = ios_base::in | ios_base::out);
    virtual int      sync();
    virtual void     imbue(const locale& loc);
  };

  template <class charT, class traits>
  void swap(basic_filebuf<charT, traits>& x,
            basic_filebuf<charT, traits>& y);
  template <class charT, class traits>
  void swap(basic_filebuf<charT, traits>&& x,
            basic_filebuf<charT, traits>& y);
  template <class charT, class traits>
  void swap(basic_filebuf<charT, traits>& x,
            basic_filebuf<charT, traits>&& y);
}
```

1 The class `basic_filebuf<charT,traits>` associates both the input sequence and the output sequence with a file.

2 The restrictions on reading and writing a sequence controlled by an object of class `basic_filebuf<charT,traits>` are the same as for reading and writing with the Standard C library `FILE`s.

3 In particular:

— If the file is not open for reading the input sequence cannot be read.

— If the file is not open for writing the output sequence cannot be written.

— A joint file position is maintained for both the input sequence and the output sequence.

4 An instance of `basic_filebuf` behaves as described in 27.8.1.1 provided `traits::pos_type` is `fpos<traits::state_type>`. Otherwise the behavior is undefined.

5 In order to support file I/O and multibyte/wide character conversion, conversions are performed using members of a facet, referred to as `a_codecvt` in following sections, obtained "as if" by

```
const codecvt<charT,char,typename traits::state_type>& a_codecvt =
  use_facet<codecvt<charT,char,typename traits::state_type> >(getloc());
```

27.8.1.2 `basic_filebuf` constructors [filebuf.cons]

```
basic_filebuf();
```

1 *Effects:* Constructs an object of class `basic_filebuf<charT,traits>`, initializing the base class with `basic_streambuf<charT,traits>()` (27.5.2.1).

2 *Postcondition:* `is_open() == false`.

```
basic_filebuf(basic_filebuf&& rhs);
```

3 *Effects:* Move constructs from the rvalue `rhs`. It is implementation-defined whether the sequence pointers in `*this` (`eback()`, `gptr()`, `egptr()`, `pbase()`, `pptr()`, `epptr()`) obtain the values which `rhs` had. Whether they do or not, `*this` and `rhs` reference separate buffers (if any at all) after the construction. Additionally `*this` references the file which `rhs` did before the construction, and `rhs` references no file after the construction. The openmode, locale and any other state of `rhs` is also copied.

4 *Postconditions:* Let `rhs_p` refer to the state of `rhs` just prior to this construction and let `rhs_a` refer to the state of `rhs` just after this construction.

- `is_open() == rhs_p.is_open()`
- `rhs_a.is_open() == false`
- `gptr() - eback() == rhs_p.gptr() - rhs_p.eback()`
- `egptr() - eback() == rhs_p.egptr() - rhs_p.eback()`
- `pptr() - pbase() == rhs_p.pptr() - rhs_p.pbase()`
- `epptr() - pbase() == rhs_p.epptr() - rhs_p.pbase()`
- `if (eback()) eback() != rhs_a.eback()`
- `if (gptr()) gptr() != rhs_a.gptr()`
- `if (egptr()) egptr() != rhs_a.egptr()`
- `if (pbase()) pbase() != rhs_a.pbase()`
- `if (pptr()) pptr() != rhs_a.pptr()`
- `if (epptr()) epptr() != rhs_a.epptr()`

```
virtual ~basic_filebuf();
```

5 *Effects:* Destroys an object of class `basic_filebuf<charT,traits>`. Calls `close()`. If an exception occurs during the destruction of the object, including the call to `close()`, the exception is caught but not rethrown (see 17.6.5.10).

27.8.1.3 Assign and swap [filebuf.assign]

```
basic_filebuf& operator=(basic_filebuf&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_filebuf&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs`.

```
template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
          basic_filebuf<charT, traits>& y);
template <class charT, class traits>
void swap(basic_filebuf<charT, traits>&& x,
          basic_filebuf<charT, traits>& y);
template <class charT, class traits>
void swap(basic_filebuf<charT, traits>& x,
          basic_filebuf<charT, traits>&& y);
```

4 *Effects:* `x.swap(y)`.

27.8.1.4 Member functions [filebuf.members]

```
bool is_open() const;
```

1 *Returns:* `true` if a previous call to `open` succeeded (returned a non-null value) and there has been no intervening call to close.

```
basic_filebuf<charT,traits>* open(const char* s,
    ios_base::openmode mode);
```

2 *Effects:* If `is_open() != false`, returns a null pointer. Otherwise, initializes the `filebuf` as required. It then opens a file, if possible, whose name is the NTBS `s` ("as if" by calling `std::fopen(s,modstr)`). The NTBS `modstr` is determined from `mode & ~ios_base::ate` as indicated in Table 109. If `mode` is not some combination of flags shown in the table then the open fails.

Table 109 — File open modes

`ios_base` flag combination					`stdio` equivalent
`binary`	`in`	`out`	`trunc`	`app`	
		+			`"w"`
		+		+	`"a"`
				+	`"a"`
		+	+		`"w"`
	+				`"r"`
	+	+			`"r+"`
	+	+	+		`"w+"`
	+	+		+	`"a+"`
	+			+	`"a+"`
+		+			`"wb"`
+		+		+	`"ab"`
+				+	`"ab"`
+		+	+		`"wb"`
+	+				`"rb"`
+	+	+			`"r+b"`
+	+	+	+		`"w+b"`
+	+	+		+	`"a+b"`
+	+			+	`"a+b"`

3 If the open operation succeeds and `(mode & ios_base::ate) != 0`, positions the file to the end ("as if" by calling `std::fseek(file,0,SEEK_END)`).[330]

4 If the repositioning operation fails, calls `close()` and returns a null pointer to indicate failure.

5 *Returns:* `this` if successful, a null pointer otherwise.

```
basic_filebuf<charT,traits>* open(const string& s,
    ios_base::openmode mode);
```

Returns: `open(s.c_str(), mode);`

330) The macro `SEEK_END` is defined, and the function signatures `fopen(const char*, const char*)` and `fseek(FILE*, long, int)` are declared, in `<cstdio>` (27.8.2).

```
basic_filebuf<charT,traits>* close();
```

6 *Effects:* If `is_open() == false`, returns a null pointer. If a put area exists, calls `overflow(traits::eof())` to flush characters. If the last virtual member function called on `*this` (between `underflow`, `overflow`, `seekoff`, and `seekpos`) was `overflow` then calls `a_codecvt.unshift` (possibly several times) to determine a termination sequence, inserts those characters and calls `overflow(traits::eof())` again. Finally, regardless of whether any of the preceding calls fails or throws an exception, the function closes the file ("as if" by calling `std::fclose(file)`).[331] If any of the calls made by the function, including `std::fclose`, fails, `close` fails by returning a null pointer. If one of these calls throws an exception, the exception is caught and rethrown after closing the file.

7 *Returns:* `this` on success, a null pointer otherwise.

8 *Postcondition:* `is_open() == false`.

27.8.1.5 Overridden virtual functions [filebuf.virtuals]

```
streamsize showmanyc();
```

1 *Effects:* Behaves the same as `basic_streambuf::showmanyc()` (27.5.2.4).

2 *Remarks:* An implementation might well provide an overriding definition for this function signature if it can determine that more characters can be read from the input sequence.

```
int_type underflow();
```

3 *Effects:* Behaves according to the description of `basic_streambuf<charT,traits>::underflow()`, with the specialization that a sequence of characters is read from the input sequence "as if" by reading from the associated file into an internal buffer (`extern_buf`) and then "as if" doing

```
char   extern_buf[XSIZE];
char*  extern_end;
charT  intern_buf[ISIZE];
charT* intern_end;
codecvt_base::result r =
  a_codecvt.in(state, extern_buf, extern_buf+XSIZE, extern_end,
               intern_buf, intern_buf+ISIZE, intern_end);
```

This shall be done in such a way that the class can recover the position (`fpos_t`) corresponding to each character between `intern_buf` and `intern_end`. If the value of `r` indicates that `a_codecvt.in()` ran out of space in `intern_buf`, retry with a larger `intern_buf`.

```
int_type uflow();
```

4 *Effects:* Behaves according to the description of `basic_streambuf<charT,traits>::uflow()`, with the specialization that a sequence of characters is read from the input with the same method as used by `underflow`.

```
int_type pbackfail(int_type c = traits::eof());
```

5 *Effects:* Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

— If `traits::eq_int_type(c,traits::eof())` returns `false` and if the function makes a putback position available and if `traits::eq(to_char_type(c),gptr()[-1])` returns `true`, decrements the next pointer for the input sequence, `gptr()`.

331) The function signature `fclose(FILE*)` is declared in `<cstdio>` (27.8.2).

Returns: `c`.

— If `traits::eq_int_type(c,traits::eof())` returns `false` and if the function makes a putback position available and if the function is permitted to assign to the putback position, decrements the next pointer for the input sequence, and stores `c` there.

Returns: `c`.

— If `traits::eq_int_type(c,traits::eof())` returns `true`, and if either the input sequence has a putback position available or the function makes a putback position available, decrements the next pointer for the input sequence, `gptr()`.

Returns: `traits::not_eof(c)`.

6 *Returns:* `traits::eof()` to indicate failure.

7 *Remarks:* If `is_open() == false`, the function always fails.

8 The function does not put back a character directly to the input sequence.

9 If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
int_type overflow(int_type c = traits::eof());
```

10 *Effects:* Behaves according to the description of `basic_streambuf<charT,traits>::overflow(c)`, except that the behavior of "consuming characters" is performed by first coverting "as if" by:

```
charT* b = pbase();
charT* p = pptr();
charT* end;
char   xbuf[XSIZE];
char*  xbuf_end;
codecvt_base::result r =
  a_codecvt.out(state, b, p, end, xbuf, xbuf+XSIZE, xbuf_end);
```

and then

— If `r == codecvt_base::error` then fail.

— If `r == codecvt_base::noconv` then output characters from `b` up to (and not including) `p`.

— If `r == codecvt_base::partial` then output to the file characters from `xbuf` up to `xbuf_end`, and repeat using characters from `end` to `p`. If output fails, fail (without repeating).

— Otherwise output from `xbuf` to `xbuf_end`, and fail if output fails. At this point if `b != p` and `b == end` (`xbuf` isn't large enough) then increase `XSIZE` and repeat from the beginning.

11 *Returns:* `traits::not_eof(c)` to indicate success, and `traits::eof()` to indicate failure. If `is_open() == false`, the function always fails.

```
basic_streambuf* setbuf(char_type* s, streamsize n);
```

12 *Effects:* If `setbuf(0,0)` is called on a stream before any I/O has occurred on that stream, the stream becomes unbuffered. Otherwise the results are implementation-defined. "Unbuffered" means that `pbase()` and `pptr()` always return null and output to the file should appear as soon as possible.

```
pos_type seekoff(off_type off, ios_base::seekdir way,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

13 *Effects:* Let `width` denote `a_codecvt.encoding()`. If `is_open() == false`, or `off != 0 && width <= 0`, then the positioning operation fails. Otherwise, if `way != basic_ios::cur` or `off != 0`, and if the last operation was output, then update the output sequence and write any unshift sequence. Next, seek to the new position: if `width > 0`, call `std::fseek(file, width * off, whence)`, otherwise call `std::fseek(file, 0, whence)`.

14 *Remarks:* "The last operation was output" means either the last virtual operation was overflow or the put buffer is non-empty. "Write any unshift sequence" means, if `width` if less than zero then call `a_codecvt.unshift(state, xbuf, xbuf+XSIZE, xbuf_end)` and output the resulting unshift sequence. The function determines one of three values for the argument `whence`, of type `int`, as indicated in Table 110.

Table 110 — `seekoff` effects

`way` Value	`stdio` Equivalent
`basic_ios::beg`	`SEEK_SET`
`basic_ios::cur`	`SEEK_CUR`
`basic_ios::end`	`SEEK_END`

15 *Returns:* a newly constructed `pos_type` object that stores the resultant stream position, if possible. If the positioning operation fails, or if the object cannot represent the resultant stream position, returns `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp,
    ios_base::openmode which = ios_base::in | ios_base::out);
```

16 Alters the file position, if possible, to correspond to the position stored in `sp` (as described below). Altering the file position performs as follows:

1. if `(om & ios_base::out) != 0`, then update the output sequence and write any unshift sequence;
2. set the file position to `sp`;
3. if `(om & ios_base::in) != 0`, then update the input sequence;

where `om` is the open mode passed to the last call to `open()`. The operation fails if `is_open()` returns false.

17 If `sp` is an invalid stream position, or if the function positions neither sequence, the positioning operation fails. If `sp` has not been obtained by a previous successful call to one of the positioning functions (`seekoff` or `seekpos`) on the same file the effects are undefined.

18 *Returns:* `sp` on success. Otherwise returns `pos_type(off_type(-1))`.

```
int sync();
```

19 *Effects:* If a put area exists, calls `filebuf::overflow` to write the characters to the file. If a get area exists, the effect is implementation-defined.

```
void imbue(const locale& loc);
```

20 *Precondition:* If the file is not positioned at its beginning and the encoding of the current locale as determined by `a_codecvt.encoding()` is state-dependent (22.2.1.4.2) then that facet is the same as the corresponding facet of `loc`.

21 *Effects:* Causes characters inserted or extracted after this call to be converted according to `loc` until another call of `imbue`.

22 *Remark:* This may require reconversion of previously converted characters. This in turn may require the implementation to be able to reconstruct the original contents of the file.

27.8.1.6 Class template `basic_ifstream` [ifstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.8.1.7 Constructors:
    basic_ifstream();
    explicit basic_ifstream(const char* s,
        ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const string& s,
        ios_base::openmode mode = ios_base::in);
    basic_ifstream(basic_ifstream&& rhs);

    // 27.8.1.8 Assign/swap:
    basic_ifstream& operator=(basic_ifstream&& rhs);
    void swap(basic_ifstream&& rhs);

    // 27.8.1.9 Members:
    basic_filebuf<charT,traits>* rdbuf() const;

    bool is_open() const;
    void open(const char* s, ios_base::openmode mode = ios_base::in);
    void open(const string& s, ios_base::openmode mode = ios_base::in);
    void close();
  private:
    // basic_filebuf<charT,traits> sb;                    exposition only
  };

  template <class charT, class traits>
  void swap(basic_ifstream<charT, traits>& x,
            basic_ifstream<charT, traits>& y);
  template <class charT, class traits>
  void swap(basic_ifstream<charT, traits>&& x,
            basic_ifstream<charT, traits>& y);
  template <class charT, class traits>
  void swap(basic_ifstream<charT, traits>& x,
            basic_ifstream<charT, traits>&& y);
}
```

1 The class `basic_ifstream<charT, traits>` supports reading from named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `filebuf` object.

27.8.1.7 `basic_ifstream` constructors [ifstream.cons]

```
basic_ifstream();
```

1 *Effects:* Constructs an object of class `basic_ifstream<charT,traits>`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.6.1.1.1, 27.8.1.2).

```
explicit basic_ifstream(const char* s,
    ios_base::openmode mode = ios_base::in);
```

2 *Effects:* Constructs an object of class `basic_ifstream`, initializing the base class with `basic_istream(&sb)` and initializing `sb` with `basic_filebuf<charT, traits>()` (27.6.1.1.1, 27.8.1.2), then calls `rdbuf()->open(s, mode | ios_base::in)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ifstream(const string& s,
    ios_base::openmode mode = ios_base::in);
```

3 *Effects:* the same as `basic_ifstream(s.c_str(), mode)`.

```
basic_ifstream(basic_ifstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.

27.8.1.8 Assign and swap [ifstream.assign]

```
basic_ifstream& operator=(basic_ifstream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_ifstream&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_istream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits>
void swap(basic_ifstream<charT, traits>& x,
          basic_ifstream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_ifstream<charT, traits>&& x,
          basic_ifstream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_ifstream<charT, traits>& x,
          basic_ifstream<charT, traits>&& y);
```

4 *Effects:* `x.swap(y)`.

27.8.1.9 Member functions [ifstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT,traits>*>(&sb)`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(const char* s, ios_base::openmode mode = ios_base::in);
```

3 *Effects:* Calls `rdbuf()->open(s, mode | ios_base::in)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

```
void open(const string& s, ios_base::openmode mode = ios_base::in);
```

4 *Effects:* calls `open(s.c_str(), mode)`.

```
void close();
```

5 *Effects:* Calls `rdbuf()->close()` and, if that function returns a null pointer, calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

27.8.1.10 Class template `basic_ofstream` [ofstream]

```
namespace std {
  template <class charT, class traits = char_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // 27.8.1.11 Constructors:
    basic_ofstream();
    explicit basic_ofstream(const char* s,
        ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const string& s,
        ios_base::openmode mode = ios_base::out);
    basic_ofstream(basic_ofstream&& rhs);

    // 27.8.1.12 Assign/swap:
    basic_ofstream& operator=(basic_ofstream&& rhs);
    void swap(basic_ofstream&& rhs);

    // 27.8.1.13 Members:
    basic_filebuf<charT,traits>* rdbuf() const;

    bool is_open() const;
    void open(const char* s, ios_base::openmode mode = ios_base::out);
    void open(const string& s, ios_base::openmode mode = ios_base::out);
    void close();
  private:
    // basic_filebuf<charT,traits> sb;                    exposition only
  };

  template <class charT, class traits>
  void swap(basic_ofstream<charT, traits>& x,
            basic_ofstream<charT, traits>& y);
  template <class charT, class traits>
```

```
    void swap(basic_ofstream<charT, traits>&& x,
              basic_ofstream<charT, traits>& y);
    template <class charT, class traits>
    void swap(basic_ofstream<charT, traits>& x,
              basic_ofstream<charT, traits>&& y);
  }
```

1 The class `basic_ofstream<charT, traits>` supports writing to named files. It uses a `basic_filebuf<charT, traits>` object to control the associated sequence. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `filebuf` object.

27.8.1.11 `basic_ofstream` constructors [ofstream.cons]

```
basic_ofstream();
```

1 *Effects:* Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.6.2.2, 27.8.1.2).

```
explicit basic_ofstream(const char* s,
    ios_base::openmode mode = ios_base::out);
```

2 *Effects:* Constructs an object of class `basic_ofstream<charT,traits>`, initializing the base class with `basic_ostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()` (27.6.2.2, 27.8.1.2), then calls `rdbuf()->open(s, mode|ios_base::out)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ofstream(const string& s,
    ios_base::openmode mode = ios_base::out);
```

3 *Effects:* the same as `basic_ofstream(s.c_str(), mode);`

```
basic_ofstream(basic_ofstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_ostream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.

27.8.1.12 Assign and swap [ofstream.assign]

```
basic_ofstream& operator=(basic_ofstream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_ofstream&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_ostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
          basic_ofstream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_ofstream<charT, traits>&& x,
```

```
          basic_ofstream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_ofstream<charT, traits>& x,
          basic_ofstream<charT, traits>&& y);
```

4 *Effects:* `x.swap(y)`.

27.8.1.13 Member functions [ofstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT,traits>*>(&sb)`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(const char* s, ios_base::openmode mode = ios_base::out);
```

3 *Effects:* Calls `rdbuf()->open(s, mode | ios_base::out)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

```
void close();
```

4 *Effects:* Calls `rdbuf()->close()` and, if that function fails (returns a null pointer), calls `setstate(failbit)` (which may throw `ios_base::failure` (27.4.4.3)).

```
void open(const string& s, ios_base::openmode mode = ios_base::out);
```

5 *Effects:* calls `open(s.c_str(), mode);`

27.8.1.14 Class template `basic_fstream` [fstream]

```
namespace std {
  template <class charT, class traits=char_traits<charT> >
  class basic_fstream
    : public basic_iostream<charT,traits> {

  public:
    typedef charT                     char_type;
    typedef typename traits::int_type int_type;
    typedef typename traits::pos_type pos_type;
    typedef typename traits::off_type off_type;
    typedef traits                    traits_type;

    // constructors/destructor
    basic_fstream();
    explicit basic_fstream(const char* s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    explicit basic_fstream(const string& s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    basic_fstream(basic_fstream&& rhs);

    // 27.8.1.16 Assign/swap:
    basic_fstream& operator=(basic_fstream&& rhs);
    void swap(basic_fstream&& rhs);
```

```
    // Members:
    basic_filebuf<charT,traits>* rdbuf() const;
    bool is_open() const;
    void open(const char* s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    void open(const string& s,
        ios_base::openmode mode = ios_base::in|ios_base::out);
    void close();

  private:
    // basic_filebuf<charT,traits> sb;                    exposition only
  };

  template <class charT, class traits>
  void swap(basic_fstream<charT, traits>& x,
            basic_fstream<charT, traits>& y);
  template <class charT, class traits>
  void swap(basic_fstream<charT, traits>&& x,
            basic_fstream<charT, traits>& y);
  template <class charT, class traits>
  void swap(basic_fstream<charT, traits>& x,
            basic_fstream<charT, traits>&& y);
}
```

1 The class template `basic_fstream<charT,traits>` supports reading and writing from named files. It uses a `basic_filebuf<charT,traits>` object to control the associated sequences. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `basic_filebuf` object.

27.8.1.15 `basic_fstream` constructors [fstream.cons]

```
basic_fstream();
```

1 *Effects:* Constructs an object of class `basic_fstream<charT,traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_filebuf<charT,traits>()`.

```
explicit basic_fstream(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

2 *Effects:* Constructs an object of class `basic_fstream<charT, traits>`, initializing the base class with `basic_iostream(&sb)` and initializing `sb` with `basic_filebuf<charT, traits>()`. Then calls `rdbuf()->open(s, mode)`. If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_fstream(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

3 *Effects:* the same as `basic_fstream(s.c_str(), mode);`

```
basic_fstream(basic_fstream&& rhs);
```

4 *Effects:* Move constructs from the rvalue `rhs`. This is accomplished by move constructing the base class, and the contained `basic_filebuf`. Next `basic_istream<charT,traits>::set_rdbuf(&sb)` is called to install the contained `basic_filebuf`.

27.8.1.16 Assign and swap [fstream.assign]

```
basic_fstream& operator=(basic_fstream&& rhs);
```

1 *Effects:* `swap(rhs)`.

2 *Returns:* `*this`.

```
void swap(basic_fstream&& rhs);
```

3 *Effects:* Exchanges the state of `*this` and `rhs` by calling `basic_iostream<charT,traits>::swap(rhs)` and `sb.swap(rhs.sb)`.

```
template <class charT, class traits>
void swap(basic_fstream<charT, traits>& x,
          basic_fstream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_fstream<charT, traits>&& x,
          basic_fstream<charT, traits>& y);
template <class charT, class traits>
void swap(basic_fstream<charT, traits>& x,
          basic_fstream<charT, traits>&& y);
```

4 *Effects:* `x.swap(y)`.

27.8.1.17 Member functions [fstream.members]

```
basic_filebuf<charT,traits>* rdbuf() const;
```

1 *Returns:* `const_cast<basic_filebuf<charT,traits>*>(&sb)`.

```
bool is_open() const;
```

2 *Returns:* `rdbuf()->is_open()`.

```
void open(const char* s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

3 *Effects:* Calls `rdbuf()->open(s,mode)`. If that function does not return a null pointer calls `clear()`, otherwise calls `setstate(failbit)`, (which may throw `ios_base::failure`) (27.4.4.3).

```
void open(const string& s,
    ios_base::openmode mode = ios_base::in|ios_base::out);
```

4 *Effects:* calls `open(s.c_str(), mode);`

```
void close();
```

5 *Effects:* Calls `rdbuf()->close()` and, if that function returns returns a null pointer, calls `setstate(failbit)` (27.4.4.3) (which may throw `ios_base::failure`).

27.8.2 C Library files [c.files]

Table 111 describes header `<cstdio>`.

See also: ISO C 7.9, Amendment 1 4.6.2.

Table 112 describes header `<cinttypes>`.

Table 111 — Header <cstdio> synopsis

Type	Name(s)				
Macros:					
BUFSIZ	FOPEN_MAX	SEEK_CUR	TMP_MAX	_IONBF	stdout
EOF	L_tmpnam	SEEK_END	_IOFBF	stderr	
FILENAME_MAX	NULL <cstdio>	SEEK_SET	_IOLBF	stdin	
Types:	FILE	fpos_t	size_t <cstdio>		
Functions:					
clearerr	fopen	fsetpos	putc	setbuf	vprintf
fclose	fprintf	ftell	putchar	setvbuf	vscanf
feof	fputc	fwrite	puts	snprintf	vsnprintf
ferror	fputs	getc	rename	sprintf	vsprintf
fflush	fread	getchar	remove	tmpfile	vsscanf
fgetc	freopen	gets	rewind	tmpnam	
fgetpos	fscanf	perror	scanf	ungetc	
fgets	fseek	printf	sscanf	vfprintf	

Table 112 — Header <cinttypes> synopsis

Type	Name(s)		
Macros:			
PRI{d i o u x X}[FAST LEAST]{8 16 32 64}			
PRI{d i o u x X}{MAX PTR}			
SCN{d i o u x X}[FAST LEAST]{8 16 32 64}			
SCN{d i o u x X}{MAX PTR}			
Types:	imaxdiv_t		
Functions:			
abs	imaxabs	strtoimax	wcstoimax
div	imaxdiv	strtoumax	wcstoumax

28 Regular expressions library [re]

1 This Clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

28.1 Definitions [re.def]

1 The following definitions shall apply to this Clause:

28.1.1 [defns.regex.collating.element]

collating element
a sequence of one or more characters within the current locale that collate as if they were a single character.

28.1.2 [defns.regex.finite.state.machine]

finite state machine
an unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

28.1.3 [defns.regex.format.specifier]

format specifier
a sequence of one or more characters that is to be replaced with some part of a regular expression match.

28.1.4 [defns.regex.matched]

matched
a sequence of zero or more characters is matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

28.1.5 [defns.regex.primary.equivalence.class]

primary equivalence class
a set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accentation, case, or locale specific tailorings.

28.1.6 [defns.regex.regular.expression]

regular expression
a pattern that selects specific strings from a set of character strings.

28.1.7 [defns.regex.subexpression]

sub-expression
a subset of a regular expression that has been marked by parenthesis.

28.2 Requirements [re.req]

1 This subclause defines requirements on classes representing regular expression traits. [*Note:* The class template `regex_traits`, defined in Clause 28.7, satisfies these requirements. — *end note*]

2 The class template `basic_regex`, defined in Clause 28.8, needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member typedefs and functions in the template parameter `traits` used by the `basic_regex` class template. This subclause defines the semantics guaranteed by these members.

3 To specialize class template `basic_regex` for a character container `CharT` and its related regular expression traits class `Traits`, use `basic_regex<CharT, Traits>`.

4 In Table 113 `X` denotes a traits class defining types and functions for the character container type `charT`; `u` is an object of type `X`; `v` is an object of type `const X`; `p` is a value of type `const charT*`; `I1` and `I2` are Input Iterators; `F1` and `F2` are forward iterators; `c` is a value of type `const charT`; `s` is an object of type `X::string_type`; `cs` is an object of type `const X::string_type`; `b` is a value of type `bool`; `I` is a value of type `int`; `cl` is an object of type `X::char_class_type`, and loc is an object of type `X::locale_type`.

Table 113 — Regular expression traits class requirements

Expression	Return type	Assertion/note pre-/post-condition
`X::char_type`	`charT`	The character container type used in the implementation of class template `basic_regex`.
`X::string_type`	`std::basic_string<charT>`	
`X::locale_type`	A copy constructible type	A type that represents the locale used by the traits class.
`X::char_class_type`	A bitmask type (17.5.3.2.3).	A bitmask type representing a particular character classification.
`X::length(p)`	`std::size_t`	Yields the smallest `i` such that `p[i] == 0`. Complexity is linear in `i` .
`v.translate(c)`	`X::char_type`	Returns a character such that for any character `d` that is to be considered equivalent to `c` then `v.translate(c) == v.translate(d)`.
`v.translate_nocase(c)`	`X::char_type`	For all characters `C` that are to be considered equivalent to `c` when comparisons are to be performed without regard to case, then `v.translate_nocase(c) == v.translate_nocase(C)`.
`v.transform(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1,F2)` such that if the character sequence `[G1,G2)` sorts before the character sequence `[H1,H2)` then `v.transform(G1, G2) < v.transform(H1, H2)`.
`v.transform_primary(F1, F2)`	`X::string_type`	Returns a sort key for the character sequence designated by the iterator range `[F1,F2)` such that if the character sequence `[G1,G2)` sorts before the character sequence `[H1,H2)` when character case is not considered then `v.transform_primary(G1, G2) < v.transform_primary(H1, H2)`.

Table 113 — Regular expression traits class requirements (continued)

Expression	Return type	Assertion/note pre-/post-condition
`v.lookup_collatename(F1, F2)`	`X::string_type`	Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range `[F1,F2)`. Returns an empty string if the character sequence is not a valid collating element.
`v.lookup_classname(F1, F2, b)`	`X::char_class_type`	Converts the character sequence designated by the iterator range `[F1,F2)` into a value of a bitmask type that can subsequently be passed to `isctype`. Values returned from `lookup_classname` can be bitwise or'ed together; the resulting value represents membership in either of the corresponding character classes. If `b` is true, the returned bitmask is suitable for matching characters without regard to their case. Returns 0 if the character sequence is not the name of a character class recognized by `X`. The value returned shall be independent of the case of the characters in the sequence.
`v.isctype(c, cl)`	`bool`	Returns `true` if character `c` is a member of one of the character classes designated by `cl`, `false` otherwise.
`v.value(c, I)`	`int`	Returns the value represented by the digit c in base I if the character c is a valid digit in base I; otherwise returns `-1`. [*Note:* the value of I will only be 8, 10, or 16. — *end note*]
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.

5 [*Note:* Class template `regex_traits` satisfies the requirements for a regular expression traits class when it is specialized for `char` or `wchar_t`. This Class template is described in the header `<regex>`, and is described in Clause 28.7. — *end note*]

28.3 Regular expressions summary [re.sum]

1 The header `<regex>` defines a basic regular expression class template and its traits that can handle all char-like template arguments (21).

2 The header `<regex>` defines a class template that holds the result of a regular expression match.

3 The header `<regex>` defines a series of algorithms that allow an iterator sequence to be operated upon by a regular expression.

4 The header `<regex>` defines two specific template classes, `regex` and `wregex`, and their special traits.

5 The header <regex> also defines two iterator types for enumerating regular expression matches.

28.4 Header <regex> synopsis [re.syn]

```
namespace std {
  // 28.5, regex constants:
  namespace regex_constants {
    typedef bitmask_type syntax_option_type;
    typedef bitmask_type match_flag_type;
    typedef implementation-defined error_type;
  } // namespace regex_constants

  // 28.6, class regex_error:
  class regex_error;

  // 28.7, class template regex_traits:
  template <class charT> struct regex_traits;

  // 28.8, class template basic_regex:
  template <class charT, class traits = regex_traits<charT> > class basic_regex;

  typedef basic_regex<char>    regex;
  typedef basic_regex<wchar_t> wregex;

  // 28.8.6, basic_regex swap:
  template <class charT, class traits>
    void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

  // 28.9, class template sub_match:
  template <class BidirectionalIterator>
    class sub_match;

  typedef sub_match<const char*>             csub_match;
  typedef sub_match<const wchar_t*>          wcsub_match;
  typedef sub_match<string::const_iterator>  ssub_match;
  typedef sub_match<wstring::const_iterator> wssub_match;

  // 28.9.2, sub_match non-member operators:
  template <class BiIter>
    bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
  template <class BiIter>
    bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);


  template <class BiIter, class ST, class SA>
    bool operator==(
      const basic_string<
        typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
```

```
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator!=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
  bool operator==(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator!=(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>=(
    const sub_match<BiIter>& lhs,
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(
    const sub_match<BiIter>& lhs,
```

```
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);

template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
```

```
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);

template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// 28.9.3, concept maps for sub_match
template<BidirectionalIterator Iter> concept_map Range<sub_match<Iter> > see below;
template<BidirectionalIterator Iter> concept_map Range<const sub_match<Iter> > see below;

// 28.10, class template match_results:
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator> > >
  class match_results;

typedef match_results<const char*>             cmatch;
typedef match_results<const wchar_t*>          wcmatch;
typedef match_results<string::const_iterator>  smatch;
typedef match_results<wstring::const_iterator> wsmatch;

// match_results comparisons
template <class BidirectionalIterator, class Allocator>
  bool operator== (const match_results<BidirectionalIterator, Allocator>& m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
  bool operator!= (const match_results<BidirectionalIterator, Allocator>& m1,
                   const match_results<BidirectionalIterator, Allocator>& m2);

// 28.10.6, match_results swap:
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);
```

```
  // 28.11.2, function template regex_match:
  template <class BidirectionalIterator, class Allocator,
      class charT, class traits>
    bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                     match_results<BidirectionalIterator, Allocator>& m,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                       regex_constants::match_default);
  template <class BidirectionalIterator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
  template <class charT, class Allocator, class traits>
    bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                       regex_constants::match_default);
  template <class ST, class SA, class Allocator, class charT, class traits>
    bool regex_match(const basic_string<charT, ST, SA>& s,
                     match_results<
                       typename basic_string<charT, ST, SA>::const_iterator,
                       Allocator>& m,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                       regex_constants::match_default);
  template <class charT, class traits>
    bool regex_match(const charT* str,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                       regex_constants::match_default);
  template <class ST, class SA, class charT, class traits>
    bool regex_match(const basic_string<charT, ST, SA>& s,
                     const basic_regex<charT, traits>& e,
                     regex_constants::match_flag_type flags =
                       regex_constants::match_default);

  // 28.11.3, function template regex_search:
  template <class BidirectionalIterator, class Allocator,
      class charT, class traits>
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                      match_results<BidirectionalIterator, Allocator>& m,
                      const basic_regex<charT, traits>& e,
                      regex_constants::match_flag_type flags =
                        regex_constants::match_default);
  template <class BidirectionalIterator, class charT, class traits>
    bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                      const basic_regex<charT, traits>& e,
                      regex_constants::match_flag_type flags =
                        regex_constants::match_default);
  template <class charT, class Allocator, class traits>
    bool regex_search(const charT* str,
                      match_results<const charT*, Allocator>& m,
                      const basic_regex<charT, traits>& e,
                      regex_constants::match_flag_type flags =
```

```
                    regex_constants::match_default);
template <class charT, class traits>
  bool regex_search(const charT* str,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<
                      typename basic_string<charT, ST, SA>::const_iterator,
                      Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);

// 28.11.4, function template regex_replace:
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT>
  OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const basic_string<charT>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
template <class traits, class charT>
  basic_string<charT>
  regex_replace(const basic_string<charT>& s,
                const basic_regex<charT, traits>& e,
                const basic_string<charT>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);

// 28.12.1, class template regex_iterator:
template <class BidirectionalIterator,
          class charT = typename iterator_traits<
            BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
  class regex_iterator;

typedef regex_iterator<const char*>             cregex_iterator;
typedef regex_iterator<const wchar_t*>          wcregex_iterator;
typedef regex_iterator<string::const_iterator>  sregex_iterator;
typedef regex_iterator<wstring::const_iterator> wsregex_iterator;

// 28.12.2, class template regex_token_iterator:
template <class BidirectionalIterator,
          class charT = typename iterator_traits<
            BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
```

```
    class regex_token_iterator;

  typedef regex_token_iterator<const char*>             cregex_token_iterator;
  typedef regex_token_iterator<const wchar_t*>          wcregex_token_iterator;
  typedef regex_token_iterator<string::const_iterator>  sregex_token_iterator;
  typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
}
```

28.5 Namespace `std::regex_constants` [re.const]

1 The namespace `std::regex_constants` holds symbolic constants used by the regular expression library. This namespace provides three types, `syntax_option_type`, `match_flag_type`, and `error_type`, along with several constants of these types.

28.5.1 Bitmask Type `syntax_option_type` [re.synopt]

```
namespace std {
  namespace regex_constants {
    typedef bitmask_type syntax_option_type;
    static const syntax_option_type icase;
    static const syntax_option_type nosubs;
    static const syntax_option_type optimize;
    static const syntax_option_type collate;
    static const syntax_option_type ECMAScript;
    static const syntax_option_type basic;
    static const syntax_option_type extended;
    static const syntax_option_type awk;
    static const syntax_option_type grep;
    static const syntax_option_type egrep;
  }
}
```

1 The type `syntax_option_type` is an implementation-defined bitmask type (17.5.3.2.3). Setting its elements has the effects listed in table 114. A valid value of type `syntax_option_type` shall have exactly one of the elements `ECMAScript`, `basic`, `extended`, `awk`, `grep`, `egrep`, set.

28.5.2 Bitmask Type `regex_constants::match_flag_type` [re.matchflag]

```
namespace std {
  namespace regex_constants{
    typedef bitmask_type match_flag_type;

    static const match_flag_type match_default = 0;
    static const match_flag_type match_not_bol;
    static const match_flag_type match_not_eol;
    static const match_flag_type match_not_bow;
    static const match_flag_type match_not_eow;
    static const match_flag_type match_any;
    static const match_flag_type match_not_null;
    static const match_flag_type match_continuous;
    static const match_flag_type match_prev_avail;
    static const match_flag_type format_default = 0;
    static const match_flag_type format_sed;
    static const match_flag_type format_no_copy;
    static const match_flag_type format_first_only;
```

Table 114 — `syntax_option_type` effects

Element	Effect(s) if set
icase	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
nosubs	Specifies that when a regular expression is matched against a character container sequence, no sub-expression matches shall be stored in the supplied match_results structure.
optimize	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output.
collate	Specifies that character ranges of the form `"[a-b]"` shall be locale sensitive.
ECMAScript	Specifies that the grammar recognized by the regular expression engine shall be that used by ECMAScript in ECMA-262, as modified in 28.13.
basic	Specifies that the grammar recognized by the regular expression engine shall be that used by basic regular expressions in POSIX, Base Definitions and Headers, Section 9, Regular Expressions.
extended	Specifies that the grammar recognized by the regular expression engine shall be that used by extended regular expressions in POSIX, Base Definitions and Headers, Section 9, Regular Expressions.
awk	Specifies that the grammar recognized by the regular expression engine shall be that used by the utility awk in POSIX.
grep	Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep in POSIX.
egrep	Specifies that the grammar recognized by the regular expression engine shall be that used by the utility grep when given the -E option in POSIX.

```
  }
}
```

1 The type `regex_constants::match_flag_type` is an implementation-defined bitmask type (17.5.3.2.3). Matching a regular expression against a sequence of characters `[first,last)` proceeds according to the rules of the grammar specified for the regular expression object, modified according to the effects listed in table 115 for any bitmask elements set.

Table 115 — `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first, last)`.

Element	Effect(s) if set
match_not_bol	The first character in the sequence `[first,last)` shall be treated as though it is not at the beginning of a line, so the character ^ in the regular expression shall not match `[first,first)`.
match_not_eol	The last character in the sequence `[first,last)` shall be treated as though it is not at the end of a line, so the character `"$"` in the regular expression shall not match `[last,last)`.
match_not_bow	The expression `"\b"` shall not match the sub-sequence `[first,first)`.
match_not_eow	The expression `"\b"` shall not match the sub-sequence `[last,last)`.

Table 115 — `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first, last)`. (continued)

Element	Effect(s) if set
match_any	If more than one match is possible then any match is an acceptable result.
match_not_null	The expression shall not match an empty sequence.
match_continuous	The expression shall only match a sub-sequence that begins at `first`.
match_prev_avail	`--first` is a valid iterator position. When this flag is set the flags match_not_bol and match_not_bow shall be ignored by the regular expression algorithms 28.11 and iterators 28.12.
format_default	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the ECMAScript replace function in ECMA-262, part 15.4.11 String.prototype.replace. In addition, during search and replace operations all non-overlapping occurrences of the regular expression shall be located and replaced, and sections of the input that did not match the expression shall be copied unchanged to the output string.
format_sed	When a regular expression match is to be replaced by a new string, the new string shall be constructed using the rules used by the sed utility in POSIX.
format_no_copy	During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression shall not be copied to the output string.
format_first_only	When specified during a search and replace operation, only the first occurrence of the regular expression shall be replaced.

28.5.3 Implementation-defined `error_type` [re.err]

```
namespace std {
  namespace regex_constants {
    typedef implementation defined error_type;

    static const error_type error_collate;
    static const error_type error_ctype;
    static const error_type error_escape;
    static const error_type error_backref;
    static const error_type error_brack;
    static const error_type error_paren;
    static const error_type error_brace;
    static const error_type error_badbrace;
    static const error_type error_range;
    static const error_type error_space;
    static const error_type error_badrepeat;
    static const error_type error_complexity;
    static const error_type error_stack;
  }
}
```

1 The type `error_type` is an implementation-defined enumeration type (17.5.3.2.2). Values of type `error_type` represent the error conditions described in table 116:

Table 116 — `error_type` values in the C locale

Value	Error condition
`error_collate`	The expression contained an invalid collating element name.
`error_ctype`	The expression contained an invalid character class name.
`error_escape`	The expression contained an invalid escaped character, or a trailing escape.
`error_backref`	The expression contained an invalid back reference.
`error_brack`	The expression contained mismatched `[` and `]`.
`error_paren`	The expression contained mismatched `(` and `)`.
`error_brace`	The expression contained mismatched `{` and `}`
`error_badbrace`	The expression contained an invalid range in a `{}` expression.
`error_range`	The expression contained an invalid character range, such as `[b-a]` in most encodings.
`error_space`	There was insufficient memory to convert the expression into a finite state machine.
`error_badrepeat`	One of `*?+{` was not preceded by a valid regular expression.
`error_complexity`	The complexity of an attempted match against a regular expression exceeded a pre-set level.
`error_stack`	There was insufficient memory to determine whether the regular expression could match the specified character sequence.

28.6 Class `regex_error` [re.badexp]

```
class regex_error : public std::runtime_error {
  public:
    explicit regex_error(regex_constants::error_type ecode);
    regex_constants::error_type code() const;
};
```

1 The class `regex_error` defines the type of objects thrown as exceptions to report errors from the regular expression library.

```
regex_error(regex_constants::error_type ecode);
```

2 *Effects:* Constructs an object of class `regex_error`.

3 *Postcondition:*: `ecode == code()`

```
regex_constants::error_type code() const;
```

4 *Returns:* The error code that was passed to the constructor.

28.7 Class template `regex_traits` [re.traits]

```
namespace std {
  template <class charT>
  struct regex_traits {
  public:
     typedef charT                          char_type;
     typedef std::basic_string<char_type> string_type;
     typedef std::locale                    locale_type;
     typedef bitmask_type                   char_class_type;
```

```
    regex_traits();
    static std::size_t length(const char_type* p);
    charT translate(charT c) const;
    charT translate_nocase(charT c) const;
    template <class ForwardIterator>
      string_type transform(ForwardIterator first, ForwardIterator last) const;
    template <class ForwardIterator>
      string_type transform_primary(
        ForwardIterator first, ForwardIterator last) const;
    template <class ForwardIterator>
      string_type lookup_collatename(
        ForwardIterator first, ForwardIterator last) const;
    template <class ForwardIterator>
      char_class_type lookup_classname(
        ForwardIterator first, ForwardIterator last, bool icase = false) const;
    bool isctype(charT c, char_class_type f) const;
    int value(charT ch, int radix) const;
    locale_type imbue(locale_type l);
    locale_type getloc()const;
  };
}
```

1 The specializations `regex_traits<char>` and `regex_traits<wchar_t>` shall be valid and shall satisfy the requirements for a regular expression traits class (28.2).

`typedef` *bitmask_type* `char_class_type;`

2 The type `char_class_type` is used to represent a character classification and is capable of holding an implementation specific set returned by `lookup_classname`.

```
static std::size_t length(const char_type* p);
```

3 *Returns:* `char_traits<charT>::length(p);`

```
charT translate(charT c) const;
```

4 *Returns:* `(c)`.

```
charT translate_nocase(charT c) const;
```

5 *Returns:* `use_facet<ctype<charT> >(getloc()).tolower(c)`.

```
template <class ForwardIterator>
  string_type transform(ForwardIterator first, ForwardIterator last) const;
```

6 *Effects:*

```
  string_type str(first, last);
  return use_facet<collate<charT> >(
    getloc()).transform(&*str.begin(), &*str.end());
```

```
template <class ForwardIterator>
  string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
```

7 *Effects:* if `typeid(use_facet<collate<charT> >) == typeid(collate_byname<charT>)` and the form of the sort key returned by `collate_byname<charT> ::transform(first, last)` is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

```
template <class ForwardIterator>
  string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
```

8 *Returns:* a sequence of one or more characters that represents the collating element consisting of the character sequence designated by the iterator range `[first,last)`. Returns an empty string if the character sequence is not a valid collating element.

```
template <class ForwardIterator>
  char_class_type lookup_classname(
    ForwardIterator first, ForwardIterator last, bool icase = false) const;
```

9 *Returns:* an unspecified value that represents the character classification named by the character sequence designated by the iterator range `[first,last)`. If the parameter `icase` is true then the returned mask identifies the character classification without regard to the case of the characters being matched, otherwise it does honor the case of the characters being matched.[332] The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

10 *Remarks:* For `regex_traits<char>`, at least the names `"d"`, `"w"`, `"s"`, `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` and `"xdigit"` shall be recognized. For `regex_traits<wchar_t>`, at least the names `L"d"`, `L"w"`, `L"s"`, `L"alnum"`, `L"alpha"`, `L"blank"`, `L"cntrl"`, `L"digit"`, `L"graph"`, `L"lower"`, `L"print"`, `L"punct"`, `L"space"`, `L"upper"` and `L"xdigit"` shall be recognized.

```
bool isctype(charT c, char_class_type f) const;
```

11 *Effects:* Determines if the character `c` is a member of the character classification represented by `f`.

12 *Returns:* Converts `f` into a value `m` of type `std::ctype_base::mask` in an unspecified manner, and returns `true` if `use_facet<ctype<charT> >(getloc()).is(c, m)` is `true`. Otherwise returns `true` if `f` bitwise or'ed with the result of calling `lookup_classname` with an iterator pair that designates the character sequence `"w"` is not equal to 0 and `c == '_'`, or if `f` bitwise or'ed with the result of calling `lookup_classname` with an iterator pair that designates the character sequence `"blank"` is not equal to 0 and `c` is one of an implementation-defined subset of the characters for which `isspace(c, getloc())` returns `true`, otherwise returns `false`.

```
int value(charT ch, int radix) const;
```

13 *Precondition:* The value of *radix* shall be 8, 10, or 16.

14 *Returns:* the value represented by the digit *ch* in base *radix* if the character *ch* is a valid digit in base *radix*; otherwise returns -1.

```
locale_type imbue(locale_type loc);
```

15 *Effects:* Imbues `this` with a copy of the locale `loc`. [*Note:* calling `imbue` with a different locale than the one currently in use invalidates all cached data held by `*this`. — *end note*]

16 *Returns:* if no locale has been previously imbued then a copy of the global locale in effect at the time of construction of `*this`, otherwise a copy of the last argument passed to `imbue`.

17 *Postcondition:* `getloc() == loc`.

```
locale_type getloc()const;
```

18 *Returns:* if no locale has been imbued then a copy of the global locale in effect at the time of construction of `*this`, otherwise a copy of the last argument passed to `imbue`.

332) For example, if the parameter `icase` is true then `[[:lower:]]` is the same as `[[:alpha:]]`.

28.8 Class template basic_regex [re.regex]

1 For a char-like type charT, specializations of class template basic_regex represent regular expressions constructed from character sequences of charT characters. In the rest of 28.8, charT denotes a given char-like type. Storage for a regular expression is allocated and freed as necessary by the member functions of class basic_regex.

2 Objects of type specialization of basic_regex are responsible for converting the sequence of charT objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions. [*Note:* implementations will typically declare some function templates as friends of basic_regex to achieve this —*end note*]

3 The functions described in this Clause report errors by throwing exceptions of type regex_error.

```
namespace std {
  template <class charT,
            class traits = regex_traits<charT> >
  class basic_regex {
  public:
    // types:
    typedef          charT                                value_type;
    typedef          regex_constants::syntax_option_type  flag_type;
    typedef typename traits::locale_type                  locale_type;

    // 28.8.1, constants:
    static const regex_constants::syntax_option_type
      icase = regex_constants::icase;
    static const regex_constants::syntax_option_type
      nosubs = regex_constants::nosubs;
    static const regex_constants::syntax_option_type
      optimize = regex_constants::optimize;
    static const regex_constants::syntax_option_type
      collate = regex_constants::collate;
    static const regex_constants::syntax_option_type
      ECMAScript = regex_constants::ECMAScript;
    static const regex_constants::syntax_option_type
      basic = regex_constants::basic;
    static const regex_constants::syntax_option_type
      extended = regex_constants::extended;
    static const regex_constants::syntax_option_type
      awk = regex_constants::awk;
    static const regex_constants::syntax_option_type
      grep = regex_constants::grep;
    static const regex_constants::syntax_option_type
      egrep = regex_constants::egrep;

    // 28.8.2, construct/copy/destroy:
    basic_regex();
    explicit basic_regex(const charT* p,
      flag_type f = regex_constants::ECMAScript);
    basic_regex(const charT* p, size_t len, flag_type f);
    basic_regex(const basic_regex&);
    template <class ST, class SA>
      explicit basic_regex(const basic_string<charT, ST, SA>& p,
                           flag_type f = regex_constants::ECMAScript);
    template <class ForwardIterator>
```

```
    basic_regex(ForwardIterator first, ForwardIterator last,
                flag_type f = regex_constants::ECMAScript);
  basic_regex(initializer_list<charT>,
    flag_type = regex_constants::ECMAScript);

  ~basic_regex();

  basic_regex& operator=(const basic_regex&);
  basic_regex& operator=(const charT* ptr);
  template <class ST, class SA>
    basic_regex& operator=(const basic_string<charT, ST, SA>& p);

  // 28.8.3, assign:
  basic_regex& assign(const basic_regex& that);
  basic_regex& assign(const charT* ptr,
    flag_type f = regex_constants::ECMAScript);
  basic_regex& assign(const charT* p, size_t len, flag_type f);
  template <class string_traits, class A>
    basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                        flag_type f = regex_constants::ECMAScript);
  template <class InputIterator>
    basic_regex& assign(InputIterator first, InputIterator last,
                        flag_type f = regex_constants::ECMAScript);
  basic_regex& assign(initializer_list<charT>,
                      flag_type = regex_constants::ECMAScript);

  // 28.8.4, const operations:
  unsigned mark_count() const;
  flag_type flags() const;

  // 28.8.5, locale:
  locale_type imbue(locale_type loc);
  locale_type getloc() const;

  // 28.8.6, swap:
  void swap(basic_regex&);
};
}
```

28.8.1 basic_regex constants [re.regex.const]

```
static const regex_constants::syntax_option_type
  icase = regex_constants::icase;
static const regex_constants::syntax_option_type
  nosubs = regex_constants::nosubs;
static const regex_constants::syntax_option_type
  optimize = regex_constants::optimize;
static const regex_constants::syntax_option_type
  collate = regex_constants::collate;
static const regex_constants::syntax_option_type
  ECMAScript = regex_constants::ECMAScript;
static const regex_constants::syntax_option_type
  basic = regex_constants::basic;
static const regex_constants::syntax_option_type
  extended = regex_constants::extended;
```

```
  static const regex_constants::syntax_option_type
    awk = regex_constants::awk;
  static const regex_constants::syntax_option_type
    grep = regex_constants::grep;
  static const regex_constants::syntax_option_type
    egrep = regex_constants::egrep;
```

1 The static constant members are provided as synonyms for the constants declared in namespace `regex_constants`.

28.8.2 `basic_regex` constructors [re.regex.construct]

```
basic_regex();
```

1 *Effects:* Constructs an object of class `basic_regex` that does not match any character sequence.

```
basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
```

2 *Requires:* p shall not be a null pointer.

3 *Throws:* `regex_error` if p is not a valid regular expression.

4 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the array of `charT` of length `char_traits<charT>::length(p)` whose first element is designated by p, and interpreted according to the flags f.

5 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(const charT* p, size_t len, flag_type f);
```

6 *Requires:* p shall not be a null pointer.

7 *Throws:* `regex_error` if p is not a valid regular expression.

8 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[p,p+len)`, and interpreted according the flags specified in f.

9 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(const basic_regex& e);
```

10 *Effects:* Constructs an object of class `basic_regex` as a copy of the object `e`.

11 *Postconditions:* `flags()` and `mark_count()` return `e.flags()` and `e.mark_count()`, respectively.

```
template <class ST, class SA>
  basic_regex(const basic_string<charT, ST, SA>& s,
              flag_type f = regex_constants::ECMAScript);
```

12 *Throws:* `regex_error` if `s` is not a valid regular expression.

13 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the string `s`, and interpreted according to the flags specified in `f`.

14 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
  basic_regex(ForwardIterator first, ForwardIterator last,
              flag_type f = regex_constants::ECMAScript);
```

15 *Throws:* `regex_error` if the sequence `[first,last)` is not a valid regular expression.

16 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[first,last)`, and interpreted according to the flags specified in `f`.

17 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex& operator=(const basic_regex& e);
```

18 *Effects:* Returns the result of `assign(e)`.

```
basic_regex& operator=(const charT* ptr);
```

19 *Requires:* `ptr` shall not be a null pointer.

20 *Effects:* Returns the result of `assign(ptr)`.

```
template <class ST, class SA>
  basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

21 *Effects:* Returns the result of `assign(p)`.

```
basic_regex(initializer_list<charT> il,
            flag_type f = regex_constants::ECMAScript);
```

22 *Effects:* Same as `basic_regex(il.begin(), il.end(), f)`.

28.8.3 `basic_regex` assign [re.regex.assign]

```
basic_regex& assign(const basic_regex& that);
```

1 *Effects:* Copies `that` into `*this` and returns `*this`.

2 *Postconditions:* `flags()` and `mark_count()` return `that.flags()` and `that.mark_count()`, respectively.

```
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);
```

3 *Returns:* `assign(string_type(ptr), f)`.

```
basic_regex& assign(const charT* ptr, size_t len,
  flag_type f = regex_constants::ECMAScript);
```

4 *Returns:* `assign(string_type(ptr, len), f)`.

```
template <class string_traits, class A>
  basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                      flag_type f = regex_constants::ECMAScript);
```

5 *Throws:* `regex_error` if `s` is not a valid regular expression.

6 *Returns:* `*this`.

7 *Effects:* Assigns the regular expression contained in the string `s`, interpreted according the flags specified in `f`. If an exception is thrown, `*this` is unchanged.

8 *Postconditions:* If no exception is thrown, `flags()` returns `f` and `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class InputIterator>
  basic_regex& assign(InputIterator first, InputIterator last,
                      flag_type f = regex_constants::ECMAScript);
```

9 *Requires:* The type `InputIterator` shall satisfy the requirements for an Input Iterator (24.1.2).

10 *Returns:* `assign(string_type(first, last), f)`.

```
basic_regex& assign(initializer_list<charT> il,
                    flag_type f = regex_constants::ECMAScript);
```

11 *Effects:* Same as `assign(il.begin(), il.end(), f)`.

12 *Returns:* `*this`.

28.8.4 `basic_regex` constant operations [re.regex.operations]

```
unsigned mark_count() const;
```

1 *Effects:* Returns the number of marked sub-expressions within the regular expression.

```
flag_type flags() const;
```

2 *Effects:* Returns a copy of the regular expression syntax flags that were passed to the object's constructor or to the last call to `assign`.

28.8.5 `basic_regex` locale [re.regex.locale]

```
locale_type imbue(locale_type loc);
```

1 *Effects:* Returns the result of `traits_inst.imbue(loc)` where `traits_inst` is a (default initialized) instance of the template type argument `traits` stored within the object. After a call to `imbue` the `basic_regex` object does not match any character sequence.

```
locale_type getloc() const;
```

2 *Effects:* Returns the result of `traits_inst.getloc()` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object.

28.8.6 `basic_regex` swap [re.regex.swap]

```
void swap(basic_regex& e);
```

1 *Effects:* Swaps the contents of the two regular expressions.

2 *Postcondition:* `*this` contains the regular expression that was in `e`, `e` contains the regular expression that was in `*this`.

3 *Complexity:* constant time.

28.8.7 `basic_regex` non-member functions [re.regex.nonmemb]

28.8.7.1 `basic_regex` non-member swap [re.regex.nmswap]

```
template <class charT, class traits>
  void swap(basic_regex<charT, traits>& lhs, basic_regex<charT, traits>& rhs);
```

1 *Effects:* Calls `lhs.swap(rhs)`.

28.9 Class template `sub_match` [re.submatch]

1 Class template `sub_match` denotes the sequence of characters matched by a particular marked sub-expression.

```
namespace std {
  template <class BidirectionalIterator>
  class sub_match : public std::pair<BidirectionalIterator, BidirectionalIterator> {
  public:
     typedef typename iterator_traits<BidirectionalIterator>::
       value_type                                          value_type;
     typedef typename iterator_traits<BidirectionalIterator>::
       difference_type                                     difference_type;
     typedef BidirectionalIterator                         iterator;

     bool matched;

     difference_type length() const;
     operator basic_string<value_type>() const;
     basic_string<value_type> str() const;

     int compare(const sub_match& s) const;
     int compare(const basic_string<value_type>& s) const;
     int compare(const value_type* s) const;
  };
}
```

28.9.1 `sub_match` members [re.submatch.members]

```
difference_type length() const;
```

1 *Returns:* `(matched ?  distance(first, second) :  0)`.

```
operator basic_string<value_type>() const;
```

2 *Returns:* `matched ? basic_string<value_type>(first, second) : basic_string<value_type>()`.

```
basic_string<value_type> str() const;
```

3 *Returns:* `matched ? basic_string<value_type>(first, second) : basic_string<value_type>()`.

```
int compare(const sub_match& s) const;
```

4 *Returns:* `str().compare(s.str())`.

```
int compare(const basic_string<value_type>& s) const;
```

5 *Returns:* `str().compare(s)`.

```
int compare(const value_type* s) const;
```

6 *Returns:* `str().compare(s)`.

28.9.2 sub_match non-member operators [re.submatch.op]

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

1 *Returns:* `lhs.compare(rhs) == 0`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

2 *Returns:* `lhs.compare(rhs) != 0`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

3 *Returns:* `lhs.compare(rhs) < 0`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

4 *Returns:* `lhs.compare(rhs) <= 0`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

5 *Returns:* `lhs.compare(rhs) >= 0`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

6 *Returns:* `lhs.compare(rhs) > 0`.

```
template <class BiIter, class ST, class SA>
  bool operator==(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

7 *Returns:* `lhs == rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator!=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

8 *Returns:* `lhs != rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator<(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

9 *Returns:* `lhs < rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator>(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
```

```
    const sub_match<BiIter>& rhs);
```

10 *Returns:* `lhs > rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator>=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

11 *Returns:* `lhs >= rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator<=(
    const basic_string<
      typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
    const sub_match<BiIter>& rhs);
```

12 *Returns:* `lhs <= rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator==(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

13 *Returns:* `lhs.str() == rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator!=(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

14 *Returns:* `lhs.str() != rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator<(const sub_match<BiIter>& lhs,
                 const basic_string<
                   typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

15 *Returns:* `lhs.str() < rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator>(const sub_match<BiIter>& lhs,
                 const basic_string<
                   typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

16 *Returns:* `lhs.str() > rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator>=(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

17 *Returns:* `lhs.str() >= rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator<=(const sub_match<BiIter>& lhs,
                  const basic_string<
                    typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

18 *Returns:* `lhs.str() <= rhs`.

```
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

19 *Returns:* `lhs == rhs.str()`.

```
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

20 *Returns:* `lhs != rhs.str()`.

```
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
```

21 *Returns:* `lhs < rhs.str()`.

```
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
```

22 *Returns:* `lhs > rhs.str()`.

```
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

23 *Returns:* `lhs >= rhs.str()`.

```
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

24 *Returns:* `lhs <= rhs.str()`.

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

25 *Returns:* `lhs.str() == rhs`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

26 *Returns:* `lhs.str() != rhs`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
```

27 *Returns:* `lhs.str() < rhs`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
```

28 *Returns:* `lhs.str() > rhs`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

29 *Returns:* `lhs.str() >= rhs`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

30 *Returns:* `lhs.str() <= rhs`.

```
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

31 *Returns:* `basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) == rhs.str()`.

```
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

32 *Returns:* `basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) != rhs.str()`.

```
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
```

33 *Returns:* `basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) < rhs.str()`.

```
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
```

34 *Returns:* `basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) > rhs.str()`.

```
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

35 *Returns:* `basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) >= rhs.str()`.

```
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

36 *Returns:* `basic_string<typename iterator_traits<BiIter>::value_type>(1, lhs) <= rhs.str()`.

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

37 *Returns:* `lhs.str() == basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs)`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

38 *Returns:* `lhs.str() != basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs)`.

```
template <class BiIter>
```

```
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
```

39 *Returns:* `lhs.str() < basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs)`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
```

40 *Returns:* `lhs.str() > basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs)`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

41 *Returns:* `lhs.str() >= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs)`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

42 *Returns:* `lhs.str() <= basic_string<typename iterator_traits<BiIter>::value_type>(1, rhs)`.

```
template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);
```

43 *Returns:* `(os << m.str())`.

28.9.3 Concept maps for sub_match [re.submatch.concepts]

```
template<BidirectionalIterator Iter>
concept_map Range<sub_match<Iter> > {
  typedef Iter iterator;
  Iter begin(sub_match<Iter>& p) { return p.first; }
  Iter end(sub_match<Iter>& p) { return p.second; }
}

template<BidirectionalIterator Iter>
concept_map Range<const sub_match<Iter> > {
  typedef Iter iterator;
  Iter begin(const sub_match<Iter>& p) { return p.first; }
  Iter end(const sub_match<Iter>& p) { return p.second; }
}
```

1 *Note:* these concept maps adapt `sub_match` to the `Range` concept.

28.10 Class template match_results [re.results]

1 Class template `match_results` denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class template `match_results`.

2 The class template `match_results` shall satisfy the requirements of a sequence container, as specified in 23.1.3, except that only operations defined for const-qualified sequence containers are supported.

3 The `sub_match` object stored at index 0 represents sub-expression 0, *i.e.* the whole match. In this case the `sub_match` member `matched` is always true. The `sub_match` object stored at index `n` denotes what matched

the marked sub-expression `n` within the matched expression. If the sub-expression `n` participated in a regular expression match then the `sub_match` member `matched` evaluates to true, and members `first` and `second` denote the range of characters `[first,second)` which formed that match. Otherwise `matched` is false, and members `first` and `second` point to the end of the sequence that was searched. [*Note:* The `sub_match` objects representing different sub-expressions that did not participate in a regular expression match need not be distinct. — *end note*]

```
namespace std {
  template <class BidirectionalIterator,
            class Allocator = allocator<sub_match<BidirectionalIterator> >
  class match_results {
  public:
     typedef sub_match<BidirectionalIterator>                  value_type;
     typedef typename Allocator::const_reference               const_reference;
     typedef const_reference                                   reference;
     typedef implementation defined                            const_iterator;
     typedef const_iterator                                    iterator;
     typedef typename iterator_traits<BidirectionalIterator>::
       difference_type                                         difference_type;
     typedef typename Allocator::size_type                     size_type;
     typedef Allocator                                         allocator_type;
     typedef typename iterator_traits<BidirectionalIterator>::
       value_type                                              char_type;
     typedef basic_string<char_type>                           string_type;

     // 28.10.1, construct/copy/destroy:
     explicit match_results(const Allocator& a = Allocator());
     match_results(const match_results& m);
     match_results& operator=(const match_results& m);
     ~match_results();

     // 28.10.2, size:
     size_type size() const;
     size_type max_size() const;
     bool empty() const;

     // 28.10.3 element access:
     difference_type length(size_type sub = 0) const;
     difference_type position(size_type sub = 0) const;
     string_type str(size_type sub = 0) const;
     const_reference operator[](size_type n) const;

     const_reference prefix() const;
     const_reference suffix() const;
     const_iterator begin() const;
     const_iterator end() const;
     const_iterator cbegin() const;
     const_iterator cend() const;

     // 28.10.4, format:
     template <class OutputIter>
       OutputIter
       format(OutputIter out,
              const string_type& fmt,
              regex_constants::match_flag_type flags =
```

```
            regex_constants::format_default) const;
    string_type
    format(const string_type& fmt,
           regex_constants::match_flag_type flags =
             regex_constants::format_default) const;

    // 28.10.5, allocator:
    allocator_type get_allocator() const;

    // 28.10.6, swap:
    void swap(match_results& that);
  };
}
```

28.10.1 `match_results` constructors [re.results.const]

1 In all match_results constructors, a copy of the Allocator argument shall be used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

```
match_results(const Allocator& a = Allocator());
```

2 *Effects:* Constructs an object of class `match_results`.

3 *Postconditions:* `size()` returns `0`. `str()` returns `basic_string<char_type>()`.

```
match_results(const match_results& m);
```

4 *Effects:* Constructs an object of class `match_results`, as a copy of `m`.

```
match_results& operator=(const match_results& m);
```

5 *Effects:* Assigns `m` to `*this`. The postconditions of this function are indicated in Table 117.

Table 117 — `match_results` assignment operator effects

Element	Value
`size()`	`m.size()`
`str(n)`	`m.str(n)` for all integers `n < m.size()`
`prefix()`	`m.prefix()`
`suffix()`	`m.suffix()`
`(*this)[n]`	`m[n]` for all integers `n < m.size()`
`length(n)`	`m.length(n)` for all integers `n < m.size()`
`position(n)`	`m.position(n)` for all integers `n < m.size()`

28.10.2 `match_results` size [re.results.size]

```
size_type size() const;
```

1 *Returns:* One plus the number of marked sub-expressions in the regular expression that was matched if `*this` represents the result of a successful match. Otherwise returns `0`. [*Note:* The state of a `match_results` object can be modified only by passing that object to `regex_match` or `regex_search`. Sections 28.11.2 and 28.11.3 specify the effects of those algorithms on their `match_results` arguments. — *end note*]

```
size_type max_size() const;
```

2 *Returns:* The maximum number of `sub_match` elements that can be stored in `*this`.

```
bool empty() const;
```

3 *Returns:* `size() == 0`.

28.10.3 `match_results` element access [re.results.acc]

```
difference_type length(size_type sub = 0) const;
```

1 *Returns:* `(*this)[sub].length()`.

```
difference_type position(size_type sub = 0) const;
```

2 *Returns:* The distance from the start of the target sequence to `(*this)[sub].first`.

```
string_type str(size_type sub = 0) const;
```

3 *Returns:* `string_type((*this)[sub])`.

```
const_reference operator[](size_type n) const;
```

4 *Returns:* A reference to the `sub_match` object representing the character sequence that matched marked sub-expression `n`. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression. If `n >= size()` then returns a `sub_match` object representing an unmatched sub-expression.

```
const_reference prefix() const;
```

5 *Returns:* A reference to the `sub_match` object representing the character sequence from the start of the string being matched/searched to the start of the match found.

```
const_reference suffix() const;
```

6 *Returns:* A reference to the `sub_match` object representing the character sequence from the end of the match found to the end of the string being matched/searched.

```
const_iterator begin() const;
const_iterator cbegin() const;
```

7 *Returns:* A starting iterator that enumerates over all the sub-expressions stored in `*this`.

```
const_iterator end() const;
const_iterator cend() const;
```

8 *Returns:* A terminating iterator that enumerates over all the sub-expressions stored in `*this`.

28.10.4 `match_results` formatting [re.results.form]

```
template <class OutputIter>
  OutputIter format(OutputIter out,
                    const string_type& fmt,
                    regex_constants::match_flag_type flags =
                      regex_constants::format_default) const;
```

1 *Requires:* The type `OutputIter` shall satisfy the requirements for an Output Iterator (24.1.3).

2 *Effects:* Copies the character sequence `[fmt.begin(),fmt.end())` to OutputIter `out`. Replaces each format specifier or escape sequence in `fmt` with either the character(s) it represents or the sequence of

characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers and escape sequences are recognized.

3 *Returns:* `out`.

```
string_type format(const string_type& fmt,
                   regex_constants::match_flag_type flags =
                     regex_constants::format_default) const;
```

4 *Effects:* Returns a copy of the string `fmt`. Replaces each format specifier or escape sequence in `fmt` with either the character(s) it represents or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers and escape sequences are recognized.

28.10.5 `match_results` allocator [re.results.all]

```
allocator_type get_allocator() const;
```

1 *Effects:* Returns a copy of the Allocator that was passed to the object's constructor.

28.10.6 `match_results` swap [re.results.swap]

```
void swap(match_results& that);
```

1 *Effects:* Swaps the contents of the two sequences.

2 *Postcondition:* `*this` contains the sequence of matched sub-expressions that were in `that`, `that` contains the sequence of matched sub-expressions that were in `*this`.

3 *Complexity:* constant time.

```
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);
```

4 *Effects:* `m1.swap(m2)`.

28.10.7 `match-results` non-member functions [re.results.nonmember]

```
template <class BidirectionalIterator, class Allocator>
bool operator==(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);
```

1 *Returns:* true only if the two objects refer to the same match.

```
template <class BidirectionalIterator, class Allocator>
bool operator!=(const match_results<BidirectionalIterator, Allocator>& m1,
                const match_results<BidirectionalIterator, Allocator>& m2);
```

2 *Returns:* `!(m1 == m2)`.

28.11 Regular expression algorithms [re.alg]

28.11.1 exceptions [re.except]

1 The algorithms described in this subclause may throw an exception of type `regex_error`. If such an

exception `e` is thrown, `e.code()` shall return either `regex_constants::error_complexity` or `regex_constants::error_stack`.

28.11.2 `regex_match` [re.alg.match]

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   match_results<BidirectionalIterator, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

1 *Requires:* The type `BidirectionalIterator` shall satisfy the requirements of a Bidirectional Iterator (24.1.5).

2 *Effects:* Determines whether there is a match between the regular expression `e`, and all of the character sequence `[first,last)`. The parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a match exists, `false` otherwise.

3 *Postconditions:* If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in table 118.

Table 118 — Effects of `regex_match` algorithm

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	false
`m.prefix().first`	first
`m.prefix().second`	first
`m.prefix().matched`	false
`m.suffix().first`	last
`m.suffix().second`	last
`m.suffix().matched`	false
`m[0].first`	first
`m[0].second`	last
`m[0].matched`	`true` if a full match was found.
`m[n].first`	For all integers `n < m.size()`, the start of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last`.
`m[n].second`	For all integers `n < m.size()`, the end of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last`.
`m[n].matched`	For all integers `n < m.size()`, `true` if sub-expression `n` participated in the match, `false` otherwise.

```
template <class BidirectionalIterator, class charT, class traits>
  bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

4 *Effects:* Behaves "as if" by constructing an instance of `match_results<BidirectionalIterator>`

what, and then returning the result of regex_match(first, last, what, e, flags).

```
template <class charT, class Allocator, class traits>
  bool regex_match(const charT* str,
                   match_results<const charT*, Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

5 *Returns:* regex_match(str, str + char_traits<charT>::length(str), m, e, flags).

```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   match_results<
                     typename basic_string<charT, ST, SA>::const_iterator,
                     Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

6 *Returns:* regex_match(s.begin(), s.end(), m, e, flags).

```
template <class charT, class traits>
  bool regex_match(const charT* str,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

7 *Returns:* regex_match(str, str + char_traits<charT>::length(str), e, flags)

```
template <class ST, class SA, class charT, class traits>
  bool regex_match(const basic_string<charT, ST, SA>& s,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags =
                     regex_constants::match_default);
```

8 *Returns:* regex_match(s.begin(), s.end(), e, flags).

28.11.3 regex_search [re.alg.search]

```
template <class BidirectionalIterator, class Allocator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                    match_results<BidirectionalIterator, Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

1 *Requires:* Type BidirectionalIterator shall satisfy the requirements of a Bidirectional Iterator (24.1.4).

2 *Effects:* Determines whether there is some sub-sequence within [first,last) that matches the regular expression e. The parameter flags is used to control how the expression is matched against the character sequence. Returns true if such a sequence exists, false otherwise.

3 *Postconditions:* If the function returns false, then the effect on parameter m is unspecified except that m.size() returns 0 and m.empty() returns true. Otherwise the effects on parameter m are given in table 119.

Table 119 — Effects of `regex_search` algorithm

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	`false`
`m.prefix().first`	`first`
`m.prefix().second`	`m[0].first`
`m.prefix().matched`	`m.prefix().first != m.prefix().second`
`m.suffix().first`	`m[0].second`
`m.suffix().second`	`last`
`m.suffix().matched`	`m.suffix().first != m.suffix().second`
`m[0].first`	The start of the sequence of characters that matched the regular expression
`m[0].second`	The end of the sequence of characters that matched the regular expression
`m[0].matched`	`true` if a match was found, and `false` otherwise.
`m[n].first`	For all integers `n < m.size()`, the start of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last`.
`m[n].second`	For all integers `n < m.size()`, the end of the sequence that matched sub-expression `n`. Alternatively, if sub-expression `n` did not participate in the match, then `last` .
`m[n].matched`	For all integers `n < m.size()`, `true` if sub-expression `n` participated in the match, `false` otherwise.

```
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags =
                    regex_constants::match_default);
```

4 *Returns:* The result of `regex_search(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    match_results<
                      typename basic_string<charT, ST, SA>::const_iterator,
                      Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

5 *Returns:* The result of `regex_search(s.begin(), s.end(), m, e, flags)`.

```
template <class BidirectionalIterator, class charT, class traits>
  bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

6 *Effects:* Behaves "as if" by constructing an object `what` of type `match_results<BidirectionalIterator>` and then returning the result of `regex_search(first, last, what, e, flags)`.

```
template <class charT, class traits>
  bool regex_search(const charT* str,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

7 *Returns:* `regex_search(str, str + char_traits<charT>::length(str), e, flags)`

```
template <class ST, class SA, class charT, class traits>
  bool regex_search(const basic_string<charT, ST, SA>& s,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags =
                      regex_constants::match_default);
```

8 *Returns:* `regex_search(s.begin(), s.end(), e, flags)`.

28.11.4 `regex_replace` [re.alg.replace]

```
template <class OutputIterator, class BidirectionalIterator,
    class traits, class charT>
  OutputIterator
  regex_replace(OutputIterator out,
                BidirectionalIterator first, BidirectionalIterator last,
                const basic_regex<charT, traits>& e,
                const basic_string<charT>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

1 *Effects:* Constructs a `regex_iterator` object `i` as if by `regex_iterator<BidirectionalIterator, charT, traits> i(first, last, e, flags)`, and uses `i` to enumerate through all of the matches `m` of type `match_results<BidirectionalIterator>` that occur within the sequence `[first,last)`. If no such matches are found and `!(flags & regex_constants ::format_no_copy)` then calls `std::copy(first, last, out)`. If any matches are found then, for each such match, if `!(flags & regex_constants::format_no_copy)` calls `std ::copy(m.prefix().first, m.prefix().second, out)`, and then calls `m.format(out, fmt, flags)`. Finally, if such a match is found and `!(flags & regex_constants ::format_no_copy)`, calls `std::copy(last_m.suffix().first, last_m.suffix().second, out)` where `last_m` is a copy of the last match found. If `flags & regex_constants::format_first_only` is non-zero then only the first match found is replaced.

2 *Returns:* `out`.

```
template <class traits, class charT>
  basic_string<charT>
  regex_replace(const basic_string<charT>& s,
                const basic_regex<charT, traits>& e,
                const basic_string<charT>& fmt,
                regex_constants::match_flag_type flags =
                  regex_constants::match_default);
```

3 *Effects:* Constructs an empty string `result` of type `basic_string<charT>`, calls `regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)`, and then returns `result`.

28.12 Regular expression Iterators [re.iter]

28.12.1 Class template regex_iterator [re.regiter]

1 The class template regex_iterator is an iterator adaptor. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A regex_iterator uses regex_search to find successive regular expression matches within the sequence from which it was constructed. After the iterator is constructed, and every time operator++ is used, the iterator finds and stores a value of match_results<BidirectionalIterator>. If the end of the sequence is reached (regex_search returns false), the iterator becomes equal to the end-of-sequence iterator value. The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>& is returned. The result of operator-> on an end-of-sequence iterator is not defined. For any other iterator value a const match_results<BidirectionalIterator>* is returned. It is impossible to store things into regex_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
namespace std {
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<
              BidirectionalIterator>::value_type,
              class traits = regex_traits<charT> >
  class regex_iterator {
  public:
     typedef basic_regex<charT, traits>           regex_type;
     typedef match_results<BidirectionalIterator> value_type;
     typedef std::ptrdiff_t                       difference_type;
     typedef const value_type*                    pointer;
     typedef const value_type&                    reference;
     typedef std::forward_iterator_tag            iterator_category;

     regex_iterator();
     regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                    const regex_type& re,
                    regex_constants::match_flag_type m =
                      regex_constants::match_default);
     regex_iterator(const regex_iterator&);
     regex_iterator& operator=(const regex_iterator&);
     bool operator==(const regex_iterator&) const;
     bool operator!=(const regex_iterator&) const;
     const value_type& operator*() const;
     const value_type* operator->() const;
     regex_iterator& operator++();
     regex_iterator operator++(int);
  private:
     // these members are shown for exposition only:
     BidirectionalIterator                begin;
     BidirectionalIterator                end;
     const regex_type*                    pregex;
     regex_constants::match_flag_type     flags;
     match_results<BidirectionalIterator> match;
  };
}
```

2 A `regex_iterator` object that is not an end-of-sequence iterator holds a *zero-length match* if `match[0].matched == true` and `match[0].first == match[0].second`. [*Note:* for example, this can occur when the part of the regular expression that matched consists only of an assertion (such as `'^'`, `'$'`, `'\b'`, `'\B'`). — *end note*]

28.12.1.1 `regex_iterator` constructors [re.regiter.cnstr]

```
regex_iterator();
```

1 *Effects:* Constructs an end-of-sequence iterator.

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               regex_constants::match_flag_type m = regex_constants::match_default);
```

2 *Effects:* Initializes `begin` and `end` to `a` and `b`, respectively, sets `pregex` to `&re`, sets `flags` to `m`, then calls `regex_search(begin, end, match, *pregex, flags)`. If this call returns `false` the constructor sets `*this` to the end-of-sequence iterator.

28.12.1.2 `regex_iterator` comparisons [re.regiter.comp]

```
bool operator==(const regex_iterator& right) const;
```

1 *Returns:* `true` if `*this` and `right` are both end-of-sequence iterators or if `begin == right.begin`, `end == right.end`, `pregex == right.pregex`, `flags == right.flags`, and `match[0] == right.match[0]`, otherwise `false`.

```
bool operator!=(const regex_iterator& right) const;
```

2 *Returns:* `!(*this == right)`.

28.12.1.3 `regex_iterator` dereference [re.regiter.deref]

```
const value_type& operator*() const;
```

1 *Returns:* `match`.

```
const value_type* operator->() const;
```

2 *Returns:* `&match`.

28.12.1.4 `regex_iterator` increment [re.regiter.incr]

```
regex_iterator& operator++();
```

1 *Effects:* Constructs a local variable `start` of type `BidirectionalIterator` and initializes it with the value of `match[0].second`.

2 If the iterator holds a zero-length match and `start == end` the operator sets `*this` to the end-of-sequence iterator and returns `*this`.

3 Otherwise, if the iterator holds a zero-length match the operator calls `regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous)`. If the call returns `true` the operator returns `*this`. Otherwise the operator increments `start` and continues as if the most recent match was not a zero-length match.

4 If the most recent match was not a zero-length match, the operator sets `flags` to `flags | regex_constants ::match_prev_avail` and calls `regex_search(start, end, match, *pregex, flags)`. If the call returns `false` the iterator sets `*this` to the end-of-sequence iterator. The iterator then returns `*this`.

5 In all cases in which the call to `regex_search` returns `true`, `match.prefix().first` shall be equal to the previous value of `match[0].second`, and for each index `i` in the half-open range `[0, match.size())` for which `match[i].matched` is true, `match[i].position()` shall return `distance(begin, match[i].first)`.

6 [*Note:* this means that `match[i].position()` gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to `regex_search`. — *end note*]

7 It is unspecified how the implementation makes these adjustments.

8 [*Note:* this means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of `regex_search` will not be called. — *end note*]

```
regex_iterator operator++(int);
```

9 *Effects:*

```
regex_iterator tmp = *this;
++(*this);
return tmp;
```

28.12.2 Class template `regex_token_iterator` [re.tokiter]

1 The class template `regex_token_iterator` is an iterator adaptor; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more sub-expressions for each match found. Each position enumerated by the iterator is a `sub_match` class template instance that represents what matched a particular sub-expression within the regular expression.

2 When class `regex_token_iterator` is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

3 After it is constructed, the iterator finds and stores a value `regex_iterator<BidirectionalIterator> position` and sets the internal count `N` to zero. It also maintains a sequence `subs` which contains a list of the sub-expressions which will be enumerated. Every time `operator++` is used the count `N` is incremented; if `N` exceeds or equals `subs.size()`, then the iterator increments member `position` and sets count `N` to zero.

4 If the end of sequence is reached (`position` is equal to the end of sequence iterator), the iterator becomes equal to the end-of-sequence iterator value, unless the sub-expression being enumerated has index -1, in which case the iterator enumerates one last sub-expression that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty sub-expression.

5 The default constructor constructs an end-of-sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>*` is returned.

6 It is impossible to store things into regex_token_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
namespace std {
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<
              BidirectionalIterator>::value_type,
              class traits = regex_traits<charT> >
  class regex_token_iterator  {
  public:
     typedef basic_regex<charT, traits>       regex_type;
     typedef sub_match<BidirectionalIterator> value_type;
     typedef std::ptrdiff_t                   difference_type;
     typedef const value_type*                pointer;
     typedef const value_type&                reference;
     typedef std::forward_iterator_tag        iterator_category;

     regex_token_iterator();
     regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                          const regex_type& re,
                          int submatch = 0,
                          regex_constants::match_flag_type m =
                            regex_constants::match_default);
     regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                          const regex_type& re,
                          const std::vector<int>& submatches,
                          regex_constants::match_flag_type m =
                            regex_constants::match_default);
     template <std::size_t N>
       regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                            const regex_type& re,
                            const int (&submatches)[N],
                            regex_constants::match_flag_type m =
                              regex_constants::match_default);
     regex_token_iterator(const regex_token_iterator&);
     regex_token_iterator& operator=(const regex_token_iterator&);
     bool operator==(const regex_token_iterator&) const;
     bool operator!=(const regex_token_iterator&) const;
     const value_type& operator*() const;
     const value_type* operator->() const;
     regex_token_iterator& operator++();
     regex_token_iterator operator++(int);
  private:  // data members for exposition only:
     typedef regex_iterator<BidirectionalIterator, charT, traits> position_iterator;
     position_iterator position;
     const value_type *result;
     value_type suffix;
     std::size_t N;
     std::vector<int> subs;
  };
}
```

7 A *suffix iterator* is a regex_token_iterator object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member result holds a pointer to the data member suffix,

the value of the member `suffix.match` is `true`, `suffix.first` points to the beginning of the final sequence, and `suffix.second` points to the end of the final sequence.

8 [*Note:* for a suffix iterator, data member `suffix.first` is the same as the end of the last match found, and `suffix.second` is the same as the end of the target sequence — *end note*]

9 The *current match* is `(*position).prefix()` if `subs[N] == -1`, or `(*position)[subs[N]]` for any other value of `subs[N]`.

28.12.2.1 `regex_token_iterator` constructors [re.tokiter.cnstr]

```
regex_token_iterator();
```

1 *Effects:* Constructs the end-of-sequence iterator.

```
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     int submatch = 0,
                     regex_constants::match_flag_type m =
                       regex_constants::match_default);

regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     const std::vector<int>& submatches,
                     regex_constants::match_flag_type m =
                       regex_constants::match_default);

template <std::size_t N>
  regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                       const regex_type& re,
                       const int (&submatches)[N],
                       regex_constants::match_flag_type m =
                         regex_constants::match_default);
```

2 *Requires:* Each of the initialization values of `submatches` shall be `>= -1`.

3 *Effects:* The first constructor initializes the member `subs` to hold the single value `submatch`. The second constructor initializes the member `subs` to hold a copy of the argument `submatches`. The third constructor initializes the member `subs` to hold a copy of the sequence of integer values pointed to by the iterator range `[&submatches,&submatches + N)`.

4 Each constructor then sets `N` to 0, and `position` to `position_iterator(a, b, re, m)`. If `position` is not an end-of-sequence iterator the constructor sets `result` to the address of the current match. Otherwise if any of the values stored in `subs` is equal to -1 the constructor sets `*this` to a suffix iterator that points to the range `[a,b)`, otherwise the constructor sets `*this` to an end-of-sequence iterator.

28.12.2.2 `regex_token_iterator` comparisons [re.tokiter.comp]

```
bool operator==(const regex_token_iterator& right) const;
```

1 *Returns:* `true` if `*this` and `right` are both end-of-sequence iterators, or if `*this` and `right` are both suffix iterators and `suffix == right.suffix`; otherwise returns `false` if `*this` or `right` is an end-of-sequence iterator or a suffix iterator. Otherwise returns `true` if `position == right.position`, `N == right.N`, and `subs == right.subs`. Otherwise returns `false`.

```
bool operator!=(const regex_token_iterator& right) const;
```

2 *Returns:* `!(*this == right)`.

28.12.2.3 `regex_token_iterator` dereference [re.tokiter.deref]

```
const value_type& operator*() const;
```

1 *Returns:* `*result`.

```
const value_type* operator->() const;
```

2 *Returns:* `result`.

28.12.2.4 `regex_token_iterator` increment [re.tokiter.incr]

```
regex_token_iterator& operator++();
```

1 *Effects:* Constructs a local variable `prev` of type `position_iterator`, initialized with the value of `position`.

2 If `*this` is a suffix iterator, sets `*this` to an end-of-sequence iterator.

3 Otherwise, if `N + 1 < subs.size()`, increments `N` and sets `result` to the address of the current match.

4 Otherwise, sets `N` to 0 and increments `position`. If `position` is not an end-of-sequence iterator the operator sets `result` to the address of the current match.

5 Otherwise, if any of the values stored in `subs` is equal to -1 and `prev->suffix().length()` is not 0 the operator sets `*this` to a suffix iterator that points to the range `[prev->suffix().first, prev->suffix().second)`.

6 Otherwise, sets `*this` to an end-of-sequence iterator.

Returns: `*this`

```
regex_token_iterator& operator++(int);
```

7 *Effects:* Constructs a copy `tmp` of `*this`, then calls `++(*this)`.

8 *Returns:* `tmp`.

28.13 Modified ECMAScript regular expression grammar [re.grammar]

1 The regular expression grammar recognized by `basic_regex` objects constructed with the ECMAScript flag is that specified by ECMA-262, except as specified below.

2 Objects of type specialization of `basic_regex` store within themselves a default-constructed instance of their `traits` template parameter, henceforth referred to as `traits_inst`. This `traits_inst` object is used to support localization of the regular expression; `basic_regex` object member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate traits member function to achieve the required effect.

3 The following productions within the ECMAScript grammar are modified as follows:

```
CharacterClass ::
  [ [lookahead ∉ {^}] ClassRanges ]
  [ ^ ClassRanges ]

ClassAtom ::
```

```
-
ClassAtomNoDash
ClassAtomExClass
ClassAtomCollatingElement
ClassAtomEquivalence
```

4 The following new productions are then added:

```
ClassAtomExClass ::
  [: ClassName :]

ClassAtomCollatingElement ::
  [. ClassName .]

ClassAtomEquivalence ::
  [= ClassName =]

ClassName ::
  ClassNameCharacter
  ClassNameCharacter ClassName

ClassNameCharacter ::
  SourceCharacter but not one of "." "=" ":"
```

5 The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in regular expressions in POSIX.

6 The regular expression grammar may be modified by any `regex_constants::syntax_option_type` flags specified when constructing an object of type specialization of `basic_regex` according to the rules in table 114.

7 A `ClassName` production, when used in `ClassAtomExClass`, is not valid if `traits_inst.lookup_classname` returns zero for that name. The names recognized as valid `ClassName`s are determined by the type of the traits class, but at least the following names shall be recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, `w`. In addition the following expressions shall be equivalent:

```
\d and [[:digit:]]

\D and [^[:digit:]]

\s and [[:space:]]

\S and [^[:space:]]

\w and [_[:alnum:]]

\W and [^_[:alnum:]]
```

8 A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string.

9 The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR'ed together and subsequently passed to `traits_inst.isctype`.

10 A `ClassName` production when used in a `ClassAtomEquivalence` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string or if the value returned by `traits_inst.transform_primary` for the result of the call to `traits_inst.lookup_collatename` is an empty string.

11 When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type `regex_error`.

12 If the *CV* of a *UnicodeEscapeSequence* is greater than the largest value that can be held in an object of type `charT` the translator shall throw an exception object of type `regex_error`. [*Note:* this means that values of the form `"uxxxx"` that do not fit in a character are invalid. — *end note*]

13 Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling `traits_inst.value`.

14 The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMA-262. The behavior is modified according to any match_flag_type flags 28.5.2 specified when using the regular expression object in one of the regular expression algorithms 28.11. The behavior is also localized by interaction with the traits class template parameter as follows:

— During matching of a regular expression finite state machine against a sequence of characters, two characters `c` and `d` are compared using the following rules:

1. if `(flags() & regex_constants::icase)` the two characters are equal if `traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d)`;
2. otherwise, if `flags() & regex_constants::collate` the two characters are equal if `traits_inst.translate(c) == traits_inst.translate(d)`;
3. otherwise, the two characters are equal if `c == d`.

— During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range `c1-c2` against a character `c` is conducted as follows: if `flags() & regex_constants ::collate` is false then the character `c` is matched if `c1 <= c && c <= c2`, otherwise `c` is matched in accordance with the following algorithm:

```
string_type str1 = string_type(1,
  flags() & icase ?
    traits_inst.translate_nocase(c1) : traits_inst.translate(c1);
string_type str2 = string_type(1,
  flags() & icase ?
    traits_inst.translate_nocase(c2) : traits_inst.translate(c2);
string_type str = string_type(1,
  flags() & icase ?
    traits_inst.translate_nocase(c) : traits_inst.translate(c);
return traits_inst.transform(str1.begin(), str1.end())
      <= traits_inst.transform(str.begin(), str.end())
  && traits_inst.transform(str.begin(), str.end())
      <= traits_inst.transform(str2.begin(), str2.end());
```

— During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using `traits::transform_primary`, and then comparing the sort keys for equality.

— During matching of a regular expression finite state machine against a sequence of characters, a character `c` is a member of a character class designated by an iterator range `[first,last)` if `traits_inst.isctype(c, traits_inst.lookup_classname(first, last, flags() & icase))` is `true`.

29 Atomic operations library [atomics]

1 This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.[333]

2 The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 120 — Atomics library summary

Subclause		Header(s)
29.1	Order and Consistency	
29.2	Lock-free Property	
29.3	Atomic Types	<cstdatomic>, <stdatomic.h>
29.4	Operations on Atomic Types	
29.5	Flag Type and Operations	

Header <cstdatomic> synopsis

```
namespace std {
  // 29.1, order and consistency
  enum memory_order;
  template <class T>
    T kill_dependency(T y);

  // 29.2, lock-free property
  #define ATOMIC_INTEGRAL_LOCK_FREE unspecified
  #define ATOMIC_ADDRESS_LOCK_FREE unspecified

  // 29.5, flag type and operations
  struct atomic_flag;
  bool atomic_flag_test_and_set(volatile atomic_flag*);
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order);
  void atomic_flag_clear(volatile atomic_flag*);
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order);

  #define ATOMIC_FLAG_INIT unspecified

  // 29.3.1, integral types
  struct atomic_bool;
  bool atomic_is_lock_free(const volatile atomic_bool*);
  void atomic_store(volatile atomic_bool*, bool);
  void atomic_store_explicit(volatile atomic_bool*, bool, memory_order);
  bool atomic_load(const volatile atomic_bool*);
  bool atomic_load_explicit(const volatile atomic_bool*, memory_order);
  bool atomic_exchange(volatile atomic_bool*);
  bool atomic_exchange_explicit(volatile atomic_bool*, bool);
  bool atomic_compare_exchange_weak(volatile atomic_bool*, bool*, bool);
  bool atomic_compare_exchange_strong(volatile atomic_bool*, bool*, bool);
```

333) Atomic objects are neither active nor radioactive.

```
bool atomic_compare_exchange_weak_explicit(volatile atomic_bool*, bool*, bool,
                                           memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_bool*, bool*, bool,
                                             memory_order, memory_order);

// For each of the integral types:
struct atomic_itype;
bool atomic_is_lock_free(const volatile atomic_itype*);
void atomic_store(volatile atomic_itype*, integral);
void atomic_store_explicit(volatile atomic_itype*, integral,
                           memory_order);
integral atomic_load(const volatile atomic_itype*);
integral atomic_load_explicit(const volatile atomic_itype*, memory_order);
integral atomic_exchange(volatile atomic_itype*, integral);
integral atomic_exchange_explicit(volatile atomic_itype*, integral,
                                  memory_order);
bool atomic_compare_exchange_weak(volatile atomic_itype*, integral*, integral);
bool atomic_compare_exchange_strong(volatile atomic_itype*, integral*, integral);
bool atomic_compare_exchange_weak_explicit(volatile atomic_itype*, integral*,
                                           integral, memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_itype*, integral*,
                                             integral, memory_order, memory_order);
integral atomic_fetch_add(volatile atomic_itype*, integral);
integral atomic_fetch_add_explicit(volatile atomic_itype*, integral,
                                   memory_order);
integral atomic_fetch_sub(volatile atomic_itype*, integral);
integral atomic_fetch_sub_explicit(volatile atomic_itype*, integral,
                                   memory_order);
integral atomic_fetch_and(volatile atomic_itype*, integral);
integral atomic_fetch_and_explicit(volatile atomic_itype*, integral,
                                   memory_order);
integral atomic_fetch_or(volatile atomic_itype*, integral);
integral atomic_fetch_or_explicit(volatile atomic_itype*, integral,
                                  memory_order);
integral atomic_fetch_xor(volatile atomic_itype*, integral);
integral atomic_fetch_xor_explicit(volatile atomic_itype*, integral,
                                   memory_order);

// 29.3.2, address types
struct atomic_address;
bool atomic_is_lock_free(const volatile atomic_address*);
void atomic_store(volatile atomic_address*, void*);
void atomic_store_explicit(volatile atomic_address*, void*, memory_order);
void* atomic_load(const volatile atomic_address*);
void* atomic_load_explicit(const volatile atomic_address*, memory_order);
void* atomic_exchange(volatile atomic_address*);
void* atomic_exchange_explicit(volatile atomic_address*, void*, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_strong(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_weak_explicit(volatile atomic_address*, void**, void*,
                                           memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_address*, void**, void*,
                                             memory_order, memory_order);
void* atomic_fetch_add(volatile atomic_address*, ptrdiff_t);
void* atomic_fetch_add_explicit(volatile atomic_address*, ptrdiff_t,
```

```
                                    memory_order);
  void* atomic_fetch_sub(volatile atomic_address*, ptrdiff_t);
  void* atomic_fetch_sub_explicit(volatile atomic_address*, ptrdiff_t,
                                    memory_order);

  // 29.3.3, generic types
  template<class T> struct atomic;
  template<class T> struct atomic<T*>;
  template<> struct atomic<integral>;

  // 29.6, fences
  void atomic_thread_fence(memory_order);
  void atomic_signal_fence(memory_order);
}
```

29.1 Order and Consistency [atomics.order]

```
namespace std {
  typedef enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, memory_order_seq_cst
  } memory_order;
}
```

1 The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization order as defined in 1.10 and may provide for operation ordering. Its enumerated values and their meanings are as follows:

- `memory_order_relaxed`: no operation orders memory.
- `memory_order_release`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a store operation performs a release operation on the affected memory location.
- `memory_order_consume`: a load operation performs a consume operation on the affected memory location.
- `memory_order_acquire`, `memory_order_acq_rel`, and `memory_order_seq_cst`: a load operation performs an acquire operation on the affected memory location.

2 There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the happens before order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation that loads a value observes either the last preceding modification according to this order S, or the result of an operation that is not `memory_order_seq_cst`. [*Note:* Although it is not explicitly required that S include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the happens before ordering. — *end note*]

3 For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M preceding X in the total order S or a later modification of M in its modification order.

4 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

5 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B,

and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

6 An atomic store shall only store a value that has been computed from constants and program input values by a finite sequence of program evaluations, such that each evaluation observes the values of variables as computed by the last prior assignment in the sequence.[334] The ordering of evaluations in this sequence shall be such that:

— if an evaluation B observes a value computed by A in a different thread, then B does not happen before A, and

— if an evaluation A is included in the sequence, then every evaluation that assigns to the same variable and happens-before A is included.

7 [*Note:* The second requirement disallows "out-of-thin-air" or "speculative" stores of atomics when relaxed atomics are used. Since unordered operations are involved, evaluations may appear in this sequence out of thread order. For example, with `x` and `y` initially zero,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(42, memory_order_relaxed);
```

is allowed to produce `r1 = r2 = 42`. The sequence of evaluations justifying this consists of:

```
y.store(42, memory_order_relaxed);
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
r2 = x.load(memory_order_relaxed);
```

On the other hand,

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);

// Thread 2:
r2 = x.load(memory_order_relaxed);
y.store(r2, memory_order_relaxed);
```

may not produce `r1 = r2 = 42`, since there is no sequence of evaluations that results in the computation of 42. In the absence of "relaxed" operations and read-modify-write operations with weaker than `memory_order_acq_rel` ordering, the second requirement has no impact. — *end note*]

8 [*Note:* The requirements do allow `r1 == r2 == 42` in the following example, with `x` and `y` initially zero:

```
// Thread 1:
r1 = x.load(memory_order_relaxed);
if (r1 == 42) y.store(r1, memory_order_relaxed);

// Thread 2:
r2 = y.load(memory_order_relaxed);
if (r2 == 42) x.store(42, memory_order_relaxed);
```

334) Among other implications, atomic variables shall not decay.

However, implementations should not allow such behavior. — *end note*]

9 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time. Implementations shall not move an atomic operation out of an unbounded loop.

```
template <class T>
  T kill_dependency(T y);
```

10 *Effects:* The argument does not carry a dependency to the return value (1.10).

11 *Returns:* `y`.

29.2 Lock-free Property [atomics.lockfree]

```
namespace std {
  #define ATOMIC_INTEGRAL_LOCK_FREE unspecified
  #define ATOMIC_ADDRESS_LOCK_FREE unspecified
}
```

1 The macros `ATOMIC_INTEGRAL_LOCK_FREE` and `ATOMIC_ADDRESS_LOCK_FREE` indicate the general lock-free property of integral and address atomic types. The properties also apply to the corresponding specializations of the `atomic` template. A value of 0 indicates that the types are never lock-free. A value of 1 indicates that the types are sometimes lock-free. A value of 2 indicates that the types are always lock-free.

2 The function `atomic_is_lock_free` (29.4) indicates whether the object is lock-free. The result of a lock-free query on one object cannot be inferred from the result of a lock-free query on another object.

3 [*Note:* Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication via memory that is mapped into a process more than once and by memory that is shared between two processes. — *end note*]

29.3 Atomic Types [atomics.types]

29.3.1 Integral Types [atomics.types.integral]

```
namespace std {
  typedef struct atomic_bool {
    bool is_lock_free() const volatile;
    void store(bool, memory_order = memory_order_seq_cst) volatile;
    bool load(memory_order = memory_order_seq_cst) const volatile;
    operator bool() const volatile;
    bool exchange(bool, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_weak(bool&, bool, memory_order, memory_order) volatile;
    bool compare_exchange_strong(bool&, bool, memory_order, memory_order) volatile;
    bool compare_exchange_weak(bool&, bool, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_weak(bool&, bool, memory_order = memory_order_seq_cst) volatile;

    atomic_bool() = default;
    constexpr atomic_bool(bool);
    atomic_bool(const atomic_bool&) = delete;
    atomic_bool& operator=(const atomic_bool&) = delete;
    bool operator=(bool) volatile;
  } atomic_bool;

  bool atomic_is_lock_free(const volatile atomic_bool*);
```

```
void atomic_store(volatile atomic_bool*, bool);
void atomic_store_explicit(volatile atomic_bool*, bool, memory_order);
bool atomic_load(const volatile atomic_bool*);
bool atomic_load_explicit(const volatile atomic_bool*, memory_order);
bool atomic_exchange(volatile atomic_bool*);
bool atomic_exchange_explicit(volatile atomic_bool*, bool);
bool atomic_compare_exchange_weak(volatile atomic_bool*, bool*, bool);
bool atomic_compare_exchange_strong(volatile atomic_bool*, bool*, bool);
bool atomic_compare_exchange_weak_explicit(volatile atomic_bool*, bool*, bool,
                                           memory_order, memory_order);
bool atomic_compare_exchange_strong_explicit(volatile atomic_bool*, bool*, bool,
                                             memory_order, memory_order);

// For each of the integral types listed below:
typedef struct atomic_itype {
  bool is_lock_free() const volatile;
  void store(integral, memory_order = memory_order_seq_cst) volatile;
  integral load(memory_order = memory_order_seq_cst) const volatile;
  operator integral() const volatile;
  integral exchange(integral,
                    memory_order = memory_order_seq_cst) volatile;
  bool compare_exchange_weak(integral&, integral,
                             memory_order, memory_order) volatile;
  bool compare_exchange_strong(integral&, integral,
                               memory_order, memory_order) volatile;
  bool compare_exchange_weak(integral&, integral,
                             memory_order = memory_order_seq_cst) volatile;
  bool compare_exchange_strong(integral&, integral,
                               memory_order = memory_order_seq_cst) volatile;
  integral fetch_add(integral,
                     memory_order = memory_order_seq_cst) volatile;
  integral fetch_sub(integral,
                     memory_order = memory_order_seq_cst) volatile;
  integral fetch_and(integral,
                     memory_order = memory_order_seq_cst) volatile;
  integral fetch_or(integral,
                     memory_order = memory_order_seq_cst) volatile;
  integral fetch_xor(integral,
                     memory_order = memory_order_seq_cst) volatile;

  atomic_itype() = default;
  constexpr atomic_itype(integral);
  atomic_itype(const atomic_itype&) = delete;
  atomic_itype& operator=(const atomic_itype &) = delete;
  integral operator=(integral) volatile;
  integral operator++(int) volatile;
  integral operator--(int) volatile;
  integral operator++() volatile;
  integral operator--() volatile;
  integral operator+=(integral) volatile;
  integral operator-=(integral) volatile;
  integral operator&=(integral) volatile;
  integral operator|=(integral) volatile;
  integral operator^=(integral) volatile;
} atomic_itype;
```

```
  bool atomic_is_lock_free(const volatile atomic_itype*);
  void atomic_store(volatile atomic_itype*, integral);
  void atomic_store_explicit(volatile atomic_itype*, integral,
                             memory_order);
  integral atomic_load(const volatile atomic_itype*);
  integral atomic_load_explicit(const volatile atomic_itype*, memory_order);
  integral atomic_exchange(volatile atomic_itype*, integral);
  integral atomic_exchange_explicit(volatile atomic_itype*, integral,
                                    memory_order);
  bool atomic_compare_exchange_weak(volatile atomic_itype*, integral*, integral);
  bool atomic_compare_exchange_strong(volatile atomic_itype*, integral*, integral);
  bool atomic_compare_exchange_weak_explicit(volatile atomic_itype*, integral*,
                                             integral, memory_order, memory_order);
  bool atomic_compare_exchange_strong_explicit(volatile atomic_itype*, integral*,
                                               integral, memory_order, memory_order);
  integral atomic_fetch_add(volatile atomic_itype*, integral);
  integral atomic_fetch_add_explicit(volatile atomic_itype*, integral,
                                     memory_order);
  integral atomic_fetch_sub(volatile atomic_itype*, integral);
  integral atomic_fetch_sub_explicit(volatile atomic_itype*, integral,
                                     memory_order);
  integral atomic_fetch_and(volatile atomic_itype*, integral);
  integral atomic_fetch_and_explicit(volatile atomic_itype*, integral,
                                     memory_order);
  integral atomic_fetch_or(volatile atomic_itype*, integral);
  integral atomic_fetch_or_explicit(volatile atomic_itype*, integral,
                                    memory_order);
  integral atomic_fetch_xor(volatile atomic_itype*, integral);
  integral atomic_fetch_xor_explicit(volatile atomic_itype*, integral,
                                     memory_order);

}
```

1 The name `atomic_`*`itype`* and the functions operating on it in the preceding synopsis are placeholders for a set of classes and functions. Throughout the preceding synopsis, `atomic_`*`itype`* should be replaced by each of the class names in table 121 and table 122, and *`integral`* should be replaced by the integral type corresponding to the class name.

2 The atomic integral types shall have standard layout. They shall each have a trivial default constructor, a constexpr value constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor. They shall each support aggregate initialization syntax.

3 The semantics of the operations on these types are defined in 29.4.

4 The `atomic_bool` type provides an atomic boolean.

5 [*Note:* The representation of atomic integral types need not have the same size as their corresponding regular types. They should have the same size whenever possible, as it eases effort required to port existing code. — *end note*]

29.3.2 Address Type [atomics.types.address]

```
namespace std {
  typedef struct atomic_address {
    bool is_lock_free() const volatile;
```

Table 121 — Atomics for built-in types

Class name	Integral type
atomic_char	char
atomic_schar	signed char
atomic_uchar	unsigned char
atomic_short	short
atomic_ushort	unsigned short
atomic_int	int
atomic_uint	unsigned int
atomic_long	long
atomic_ulong	unsigned long
atomic_llong	long long
atomic_ullong	unsigned long long
atomic_char16_t	char16_t
atomic_char32_t	char32_t
atomic_wchar_t	wchar_t

```
  void store(void*, memory_order = memory_order_seq_cst) volatile;
  void* load(memory_order = memory_order_seq_cst) const volatile;
  operator void*() const volatile;
  void* exchange(void*, memory_order = memory_order_seq_cst) volatile;
  bool compare_exchange_weak(void*&, void*,
                             memory_order, memory_order) volatile;
  bool compare_exchange_strong(void*&, void*,
                               memory_order, memory_order) volatile;
  bool compare_exchange_weak(void*&, void*,
                             memory_order = memory_order_seq_cst) volatile;
  bool compare_exchange_strong(void*&, void*,
                               memory_order = memory_order_seq_cst) volatile;
  void* fetch_add(ptrdiff_t,
                  memory_order = memory_order_seq_cst) volatile;
  void* fetch_sub(ptrdiff_t,
                  memory_order = memory_order_seq_cst) volatile;

  atomic_address() = default;
  constexpr atomic_address(void*);
  atomic_address(const atomic_address&) = delete;
  atomic_address& operator=(const atomic_address&) = delete;
  void* operator=(void*) volatile;
  void* operator+=(ptrdiff_t) volatile;
  void* operator-=(ptrdiff_t) volatile;
} atomic_address;

bool atomic_is_lock_free(const volatile atomic_address*);
void atomic_store(volatile atomic_address*, void*);
void atomic_store_explicit(volatile atomic_address*, void*, memory_order);
void* atomic_load(const volatile atomic_address*);
void* atomic_load_explicit(const volatile atomic_address*, memory_order);
void* atomic_exchange(volatile atomic_address*);
void* atomic_exchange_explicit(volatile atomic_address*, void*, memory_order);
bool atomic_compare_exchange_weak(volatile atomic_address*, void**, void*);
bool atomic_compare_exchange_strong(volatile atomic_address*, void**, void*);
```

Table 122 — Atomics for standard typedef types

Class name	Integral type
atomic_int_least8_t	int_least8_t
atomic_uint_least8_t	uint_least8_t
atomic_int_least16_t	int_least16_t
atomic_uint_least16_t	uint_least16_t
atomic_int_least32_t	int_least32_t
atomic_uint_least32_t	uint_least32_t
atomic_int_least64_t	int_least64_t
atomic_uint_least64_t	uint_least64_t
atomic_int_fast8_t	int_fast8_t
atomic_uint_fast8_t	uint_fast8_t
atomic_int_fast16_t	int_fast16_t
atomic_uint_fast16_t	uint_fast16_t
atomic_int_fast32_t	int_fast32_t
atomic_uint_fast32_t	uint_fast32_t
atomic_int_fast64_t	int_fast64_t
atomic_uint_fast64_t	uint_fast64_t
atomic_intptr_t	intptr_t
atomic_uintptr_t	uintptr_t
atomic_size_t	size_t
atomic_ssize_t	ssize_t
atomic_ptrdiff_t	ptrdiff_t
atomic_intmax_t	intmax_t
atomic_uintmax_t	uintmax_t

```
  bool atomic_compare_exchange_weak_explicit(volatile atomic_address*, void**, void*,
                                             memory_order, memory_order);
  bool atomic_compare_exchange_strong_explicit(volatile atomic_address*, void**, void*,
                                               memory_order, memory_order);
  void* atomic_fetch_add(volatile atomic_address*, ptrdiff_t);
  void* atomic_fetch_add_explicit(volatile atomic_address*, ptrdiff_t,
                                  memory_order);
  void* atomic_fetch_sub(volatile atomic_address*, ptrdiff_t);
  void* atomic_fetch_sub_explicit(volatile atomic_address*, ptrdiff_t,
                                  memory_order);
}
```

1 The type `atomic_address` shall have standard layout. It shall have a trivial default constructor, a constexpr value constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor. It shall support aggregate initialization syntax.

2 The semantics of the operations on this type are defined in 29.4.

3 The `atomic_address` type provides atomic `void*` operations. The unit of addition/subtraction shall be one byte.

4 [*Note:* The representation of the atomic address type need not have the same size as its corresponding regular type. It should have the same size whenever possible, as it eases effort required to port existing

code. — *end note*]

29.3.3 Generic Types [atomics.types.generic]

```
namespace std {
  template <class T> struct atomic {
    bool is_lock_free() const volatile;
    void store(T, memory_order = memory_order_seq_cst) volatile;
    T load(memory_order = memory_order_seq_cst) const volatile;
    operator T() const volatile;
    T exchange(T, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile;
    bool compare_exchange_weak(T&, T, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_strong(T&, T, memory_order = memory_order_seq_cst) volatile;

    atomic() = default;
    constexpr atomic(T);
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    T operator=(T) volatile;
  };

  template <> struct atomic<integral> : atomic_itype {
    atomic() = default;
    constexpr atomic(integral);
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    integral operator=(integral) volatile;
    operator integral() const volatile;
  };

  template <class T> struct atomic<T*> : atomic_address {
    void store(T*, memory_order = memory_order_seq_cst) volatile;
    T* load(memory_order = memory_order_seq_cst) const volatile;
    operator T*() const volatile;
    T* exchange(T*, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_weak(T*&, T*, memory_order, memory_order) volatile;
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) volatile;
    bool compare_exchange_weak(T*&, T*, memory_order = memory_order_seq_cst) volatile;
    bool compare_exchange_strong(T*&, T*, memory_order = memory_order_seq_cst) volatile;
    T* fetch_add(ptrdiff_t, memory_order = memory_order_seq_cst) volatile;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order_seq_cst) volatile;

    atomic() = default;
    constexpr atomic(T*);
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;

    T* operator=(T*) volatile;
    T* operator++(int) volatile;
    T* operator--(int) volatile;
    T* operator++() volatile;
    T* operator--() volatile;
    T* operator+=(ptrdiff_t) volatile;
    T* operator-=(ptrdiff_t) volatile;
```

```
  };
}
```

1 There is a generic class template `atomic<T>`. The type of the template argument `T` shall be trivially copy assignable and bitwise equality comparable. [*Note:* Type arguments that are not also statically initializable and trivially destructable may be difficult to use. — *end note*]

2 Specializations of the `atomic` template shall have a deleted copy constructor, a deleted copy assignment operator, and a constexpr value constructor.

3 There are full specializations over the integral types on the `atomic` class template. For each integral type `integral` in the second column of table 121 or table 122, the specialization `atomic<integral>` shall be publicly derived from the corresponding atomic integral type in the first column of the table. These specializations shall have trivial default constructors and trivial destructors.

4 There are pointer partial specializations on the `atomic` class template. These specializations shall be publicly derived from `atomic_address`. The unit of addition/subtraction for these specializations shall be the size of the referenced type. These specializations shall have trivial default constructors and trivial destructors.

29.4 Operations on Atomic Types [atomics.types.operations]

1 There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.3.1, 29.3.2, and 29.3.3.

2 In the following operation definitions:

- an *A* refers to one of the atomic types
- a *C* refers to its corresponding non-atomic type. The `atomic_address` atomic type corresponds to the `void*` non-atomic type
- an *M* refers to type of the other argument for arithmetic operations. For integral atomic types, *M* is *C*. For atomic address types, *M* is `std::ptrdiff_t`
- the free functions not ending in `_explicit` have the semantics of their corresponding `_explicit` with `memory_order` arguments of `memory_order_seq_cst`.

3 [*Note:* Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. Thus, volatile qualified operations on non-volatile objects may be merged under some conditions. — *end note*]

```
constexpr A::A(C desired);
```

4 *Effects:* Initializes the object with the value `desired`. [*Note:* Construction is not atomic. — *end note*]

```
bool atomic_is_lock_free(const volatile A *object);
void A::is_lock_free() const volatile;
```

5 *Returns:* True if the object's operations are lock-free, false otherwise.

```
void atomic_store(volatile A *object, C desired);
void atomic_store_explicit(volatile A *object, C desired, memory_order order);
void A::store(C desired, memory_order order = memory_order_seq_cst) volatile;
```

6 *Requires:* The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

7 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the value of `desired`. Memory is affected according to the value of `order`.

```
C A::operator=(C desired) volatile;
```

8 *Effects:* `store(desired)`

9 *Returns:* `desired`

```
C atomic_load(const volatile A* object);
C atomic_load_explicit(const volatile A* object, memory_order);
C A::load(memory_order order = memory_order_seq_cst) const volatile;
```

10 *Requires:* The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

11 *Effects:* Memory is affected according to the value of `order`.

12 *Returns:* Atomically returns the value pointed to by `object` or by `this`.

```
A::operator C() const volatile;
```

13 *Effects:* `load()`

14 *Returns:* the result of `load()`.

```
C atomic_exchange(volatile A* object, C desired);
C atomic_exchange_explicit(volatile A* object, C desired, memory_order);
C A::exchange(C desired, memory_order order = memory_order_seq_cst) volatile;
```

15 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with `desired`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

16 *Returns:* Atomically returns the value pointed to by `object` or by `this` immediately before the effects.

```
bool atomic_compare_exchange_weak(volatile A* object, C* expected, C desired);
bool atomic_compare_exchange_strong(volatile A* object, C* expected, C desired);
bool atomic_compare_exchange_weak_explicit(volatile A* object, C* expected, C desired,
    memory_order success, memory_order failure);
bool atomic_compare_exchange_strong_explicit(volatile A* object, C* expected, C desired,
    memory_order success, memory_order failure);
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order success, memory_order failure ) volatile;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order success, memory_order failure ) volatile;
bool A::compare_exchange_weak(C& expected, C desired,
    memory_order order = memory_order_seq_cst) volatile;
bool A::compare_exchange_strong(C& expected, C desired,
    memory_order order = memory_order_seq_cst) volatile;
```

17 *Requires:* The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`. The `failure` argument shall be no stronger than the `success` argument.

18 *Effects:* Atomically, compares the value pointed to by `object` or by `this` for equality with that in `expected`, and if true, replaces the value pointed to by `object` or by `this` with `desired`, and if false, updates the value in `expected` with the value pointed to by `object` or by `this`. Further, if the comparison is true, memory is affected according to the value of `success`, and if the comparison is false, memory is affected according to the value of `failure`. When only one `memory_order` argument is supplied, the value of `success` is `order`, and the value of `failure` is `order` except that a

value of `memory_order_acq_rel` shall be replaced by the value `memory_order_acquire` and a value of `memory_order_release` shall be replaced by the value `memory_order_relaxed`. These operations are atomic read-modify-write operations (1.10).

19 *Returns:* The result of the comparison.

20 [*Note:* The effect of the compare-and-exchange operations is

```
if (*object == *expected)
    *object = desired;
else
    *expected = *object;
```

— *end note*]

21 *Remark:* The weak compare-and-exchange operations may fail spuriously, that is, return false while leaving the value pointed to by `expected` unchanged. [*Note:* This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g. load-locked store-conditional machines. — *end note*] [*Example:* A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop.

```
expected = current.load();
do desired = function(expected);
while (!current.compare_exchange(expected, desired));
```

When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

— *end example*]

22 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 123 — Atomic arithmetic computations

Key	Op	Computation	Key	Op	Computation
`add`	`+`	addition	`sub`	`-`	subtraction
`or`	`|`	bitwise inclusive or	`xor`	`^`	bitwise exclusive or
`and`	`&`	bitwise and			

```
C atomic_fetch_key(volatile A *object, M operand);
C atomic_fetch_key_explicit(volatile A *object, M operand, memory_order order);
C A::fetch_key(M operand, memory_order order = memory_order_seq_cst) volatile;
```

23 *Effects:* Atomically replaces the value pointed to by `object` or by `this` with the result of the *computation* applied to the value pointed to by `object` or by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

24 *Returns:* Atomically, the value pointed to by `object` or by `this` immediately before the effects.

25 *Remark:* For signed integral types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

```
C A::operator op=(M operand) volatile;
```

26 *Effects:* fetch_*key*(operand)

Returns: fetch_*key*(operand) op operand

C A::operator++(int) volatile;

27 *Returns:* fetch_add(1)

C A::operator--(int) volatile;

28 *Returns:* fetch_sub(1)

C A::operator++() volatile;

29 *Effects:* fetch_add(1)

30 *Returns:* fetch_add(1) + 1

C A::operator--() volatile;

31 *Effects:* fetch_sub(1)

32 *Returns:* fetch_sub(1) - 1

29.5 Flag Type and Operations [atomics.flag]

```
namespace std {
  typedef struct atomic_flag {
    bool test_and_set(memory_order = memory_order_seq_cst) volatile;
    void clear(memory_order = memory_order_seq_cst) volatile;

    atomic_flag() = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
  } atomic_flag;

  bool atomic_flag_test_and_set(volatile atomic_flag*);
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order);
  void atomic_flag_clear(volatile atomic_flag*);
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order);

  #define ATOMIC_FLAG_INIT unspecified
}
```

1 The atomic_flag type provides the classic test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type atomic_flag shall be lock-free. [*Note:* Hence the operations should also be address-free. No other type requires lock-free operations, so the atomic_flag type is the minimum hardware-implemented type needed to conform to this International standard. The remaining types can be emulated with atomic_flag, though with less than ideal properties. —*end note*]

3 The atomic_flag type shall have standard layout. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, and a trivial destructor.

4 The macro ATOMIC_FLAG_INIT shall be defined in such a way that it can be used to initialize an object of type atomic_flag to the clear state. For a static-duration object, that initialization shall be static. A program that uses an object of type atomic_flag without initializing it with the macro ATOMIC_FLAG_INIT is ill-formed.[*Example:*

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

— *end example*]

```
bool atomic_flag_test_and_set(volatile atomic_flag *object);
bool atomic_flag_test_and_set_explicit(volatile atomic_flag *object, memory_order order);
bool atomic_flag::test_and_set(memory_order order = memory_order_seq_cst) volatile;
```

5 *Effects:* Atomically sets the value pointed to by `object` or by `this` to true. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (1.10).

6 *Returns:* Atomically, the value of the object immediately before the effects.

```
void atomic_flag_clear(volatile atomic_flag *object);
void atomic_flag_clear_explicit(volatile atomic_flag *object, memory_order order);
void atomic_flag::clear(memory_order order = memory_order_seq_cst) volatile;
```

7 *Requires:* The `order` argument shall not be `memory_order_acquire` nor `memory_order_acq_rel`.

8 *Effects:* Atomically sets the value pointed to by `object` or by `this` to false. Memory is affected according to the value of `order`.

29.6 Fences [atomics.fences]

1 This section introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*. A fence with release semantics is called a *release fence*.

2 A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

3 A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

4 An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

```
void atomic_thread_fence(memory_order order);
```

5 *Effects:* depending on the value of `order`, this operation:

- has no effects, if `order == memory_order_relaxed`;
- is an acquire fence, if `order == memory_order_acquire || order == memory_order_consume`;
- is a release fence, if `order == memory_order_release`;
- is both an acquire fence and a release fence, if `order == memory_order_acq_rel`;
- is a sequentially consistent acquire and release fence, if `order == memory_order_seq_cst`.

```
void atomic_signal_fence(memory_order order);
```

6 *Effects:* equivalent to `atomic_thread_fence(order)`, except that synchronizes with relationships are established only between a thread and a signal handler executed in the same thread.

7 *Note:* `atomic_signal_fence` can be used to specify the order in which actions performed by the thread become visible to the signal handler.

8 *Note:* compiler optimizations and reorderings of loads and stores are inhibited in the same way as with `atomic_thread_fence`, but the hardware fence instructions that `atomic_thread_fence` would have inserted are not emitted.

30 Thread support library [thread]

1 The following subclauses describe components to create and manage threads (1.10), perform mutual exclusion, and communicate conditions between threads, as summarized in Table 124.

Table 124 — Thread support library summary

	Subclause	Header(s)
30.2	Threads	`<thread>`
30.3	Mutual exclusion	`<mutex>`
30.4	Condition variables	`<condition_variable>`
30.5	Futures	`<future>`

30.1 Requirements [thread.req]

30.1.1 Template parameter names [thread.req.paramname]

1 Throughout this Clause, the names of template parameters are used to express type requirements.

2 If a parameter is `Predicate`, `operator()` applied to the actual template argument shall return a value that is convertible to `bool`.

30.1.2 Exceptions [thread.req.exception]

1 Implementations of functions described in this Clause are permitted to call operating system or other low-level applications program interfaces (API's). Some functions described in this Clause are specified to throw exceptions of type system_error (19.4.5). Such exceptions shall be thrown if such a call results in an error that prevents the library function from satisfying its postconditions or from returning a meaningful value.

2 The error_category (19.4.1.1) of the `error_code` reported by such an exception's `code()` member function is as specified in the error condition Clause.

30.1.3 Native handles [thread.req.native]

1 Several classes described in this Clause have members `native_handle_type` and `native_handle`. The presence of these members and their semantics is implementation defined. [*Note:* These members allow implementations to provide access to implementation details. Their names are specified to facilitate portable compile-time detection. Actual use of these members is inherently non-portable. — *end note*]

30.1.4 Timing specifications [thread.req.timing]

1 Several functions described in this Clause take an argument to specify a timeout. These timeouts are specified as either a `duration` or a `time_point` type as specified in (20.8).

2 The member functions whose names end in `_for` take an argument that specifies a relative time. Implementations should use a monotonic clock to measure time for these functions.

3 The resolution of timing provided by an implementation depends on both operating system and hardware. The finest resolution provided by an implementation is called the *native resolution*.

30.2 Threads [thread.threads]

1 30.2 describes components that can be used to create and manage threads. [*Note:* These threads are intended to map one-to-one with operating system threads. — *end note*]

Header <thread> synopsis

```
namespace std {
  class thread;

  void swap(thread& x, thread& y);
  void swap(thread&& x, thread& y);
  void swap(thread& x, thread&& y);

  namespace this_thread {
  thread::id get_id();

  void yield();
  template <class Clock, class Duration>
    void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
  template <class Rep, class Period>
    void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

30.2.1 Class thread [thread.thread.class]

1 The class `thread` provides a mechanism to create a new thread of execution, to join with a thread (i.e. wait for a thread to complete), and to perform other operations that manage and query the state of a thread. A `thread` object uniquely represents a particular thread of execution. That representation may be transferred to other `thread` objects in such a way that no two `thread` objects simultaneously represent the same thread of execution. A thread of execution is *detached* when no `thread` object represents that thread. Objects of class `thread` can be in a state that does not represent a thread of execution. [*Note:* A `thread` object does not represent a thread of execution after default construction, after being moved from, or after a successful call to `detach` or `join`. — *end note*]

```
namespace std {
  class thread {
  public:
    // types:
    class id;
    typedef implementation-defined native_handle_type; // See 30.1.3

    // construct/copy/destroy:
    thread();
    template <class F> explicit thread(F f);
    template <class F, class ...Args> thread(F&& f, Args&&... args);
    ~thread();
    thread(const thread&) = delete;
    thread(thread&&);
    thread& operator=(const thread&) = delete;
    thread& operator=(thread&&);
```

```
    // members:
    void swap(thread&&);
    bool joinable() const;
    void join();
    void detach();
    id get_id() const;
    native_handle_type native_handle(); // See 30.1.3

    // static members:
    static unsigned hardware_concurrency();
  };
}
```

30.2.1.1 Class `thread::id` [thread.thread.id]

```
namespace std {
  class thread::id {
  public:
      id();
  };

  bool operator==(thread::id x, thread::id y);
  bool operator!=(thread::id x, thread::id y);
  bool operator<(thread::id x, thread::id y);
  bool operator<=(thread::id x, thread::id y);
  bool operator>(thread::id x, thread::id y);
  bool operator>=(thread::id x, thread::id y);

  template<class charT, class traits>
    basic_ostream<charT, traits>&
      operator<< (basic_ostream<charT, traits>&& out, thread::id id);
}
```

1 An object of type `thread::id` provides a unique identifier for each thread of execution and a single distinct value for all `thread` objects that do not represent a thread of execution (30.2.1). Each thread of execution has an associated `thread::id` object that is not equal to the `thread::id` object of any other thread of execution and that is not equal to the `thread::id` object of any `std::thread` object that does not represent threads of execution. The library may reuse the value of a `thread::id` of a terminated thread that can no longer be joined.

2 [*Note:* Relational operators allow `thread::id` objects to be used as keys in associative containers. — *end note*]

`id();`

3 *Effects:* Constructs an object of type `id`.

4 *Throws:* Nothing.

5 *Postconditions:* The constructed object does not represent a thread of execution.

`bool operator==(thread::id x, thread::id y).`

6 *Returns:* `true` only if `x` and `y` represent the same thread of execution or neither `x` nor `y` represents a thread of execution.

7 *Throws:* Nothing.

```
bool operator!=(thread::id x, thread::id y);
```

8 *Returns:* `!(x == y)`

9 *Throws:* Nothing.

```
bool operator<(thread::id x, thread::id y);
```

10 *Returns:* A value such that `operator<` is a total ordering as described in 25.3.

11 *Throws:* Nothing.

```
bool operator<=(thread::id x, thread::id y);
```

12 *Returns:* `!(y < x)`

13 *Throws:* Nothing.

```
bool operator>(thread::id x, thread::id y);
```

14 *Returns:* `y < x`

15 *Throws:* Nothing.

```
bool operator>=(thread::id x, thread::id y);
```

16 *Returns:* `!(x < y)`

17 *Throws:* Nothing.

```
template<class charT, class traits>
  basic_ostream<charT, traits>&
    operator<< (basic_ostream<charT, traits>&& out, thread::id id);
```

18 *Effects:* Inserts an unspecified text representation of `id` into `out`. For two objects of type `thread::id` `x` and `y`, if `x == y` the `thread::id` objects shall have the same text representation and if `x != y` the `thread::id` objects shall have distinct text representations.

19 *Returns:* `out`

30.2.1.2 `thread` constructors [thread.thread.constr]

```
thread();
```

1 *Effects:* Constructs a `thread` object that does not represent a thread of execution.

2 *Postcondition:* `get_id() == id()`

3 *Throws:* Nothing.

```
template <class F> explicit thread(F f);
template <class F, class ...Args> thread(F&& f, Args&&... args);
```

4 *Requires:* `F` and each `Ti` in `Args` shall be `CopyConstructible` if an lvalue and otherwise `MoveConstructible`. *`INVOKE`*`(f, w1, w2, ..., wN)` (20.6.2) shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(Args)`.

5 *Effects:* Constructs an object of type `thread` and executes *`INVOKE`*`(f, t1, t2, ..., tN)` in a new thread of execution, where `t1, t2, ..., tN` are the values in `args...`. Any return value from `f` is ignored. If `f` terminates with an uncaught exception, `std::terminate()` shall be called.

6 *Synchronization:* The invocation of the constructor *happens before* the invocation of `f`.

7 *Postconditions:* `get_id() != id()`. `*this` represents the newly started thread.

8 *Throws:* `std::system_error` if unable to start the new thread.

9 *Error conditions:*

— `resource_unavailable_try_again` — the system lacked the necessary resources to create another thread, or the system-imposed limit on the number of threads in a process would be exceeded.

```
thread(thread&& x);
```

10 *Effects:* Constructs an object of type `thread` from `x`, and sets `x` to a default constructed state.

11 *Postconditions:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the start of construction.

12 *Throws:* Nothing.

30.2.1.3 `thread` destructor [thread.thread.destr]

```
~thread();
```

1 *Effects:* If `joinable()` then `detach()`, otherwise no effects. [*Note:* Destroying a joinable thread can be unsafe if the thread accesses objects or the standard library unless the thread performs explicit synchronization to ensure that it does not access the objects or the standard library past their respective lifetimes. Terminating the process with `_exit` or `quick_exit` removes some of these obligations. — *end note*]

2 *Throws:* Nothing.

30.2.1.4 `thread` assignment [thread.thread.assign]

```
thread& operator=(thread&& x);
```

1 *Effects:* If `joinable()`, calls `detach()`. Then assigns the state of `x` to `*this` and sets `x` to a default constructed state.

2 *Postconditions:* `x.get_id() == id()` and `get_id()` returns the value of `x.get_id()` prior to the assignment.

3 *Throws:* Nothing.

30.2.1.5 `thread` members [thread.thread.member]

```
void swap(thread&& x);
```

1 *Effects:* Swaps the state of `*this` and `x`.

2 *Throws:* Nothing.

```
bool joinable() const;
```

3 *Returns:* `get_id() != id()`

4 *Throws:* Nothing.

```
void join();
```

Precondition: `joinable()` is `true`.

Synchronization: The completion of the thread represented by `*this` *happens before* (1.10) `join()` returns. [*Note:* Operations on `*this` are not synchronized. — *end note*]

Postconditions: If `join()` throws an exception, the value returned by `get_id()` is unchanged. Otherwise, `get_id() == id()`.

Throws: `std::system_error` when the postconditions cannot be achieved.

Error conditions:

- `resource_deadlock_would_occur` — if deadlock is detected or `this->get_id()` == std::this_thread::get_id().
- `no_such_process` — if the thread is not valid.
- `invalid_argument` — if the thread is not joinable.

```
void detach();
```

Precondition: `joinable()` is `true`.

Effects: The thread represented by `*this` continues execution without the calling thread blocking. When `detach()` returns, `*this` no longer represents the possibly continuing thread of execution. When the thread previously represented by `*this` ends execution, the implementation shall release any owned resources.

Postcondition: `get_id() == id()`.

Throws: `std::system_error` when the effects or postconditions cannot be achieved.

Error conditions:

- `no_such_process` — not a valid thread.
- `invalid_argument` — not a detachable thread.

```
id get_id() const;
```

Returns: A default constructed `id` object if `*this` does not represent a thread, otherwise `this_thread::get_id()` for the thread of execution represented by `*this`.

Throws: Nothing.

30.2.1.6 `thread` static members [thread.thread.static]

```
unsigned hardware_concurrency();
```

Returns: The number of hardware thread contexts. [*Note:* This value should only be considered to be a hint. — *end note*] If this value is not computable or well defined an implementation should return 0.

30.2.1.7 `thread` specialized algorithms [thread.thread.algorithm]

```
void swap(thread& x, thread& y);
void swap(thread&& x, thread& y);
void swap(thread& x, thread&& y);
```

Effects: `x.swap(y)`

30.2.2 Namespace this_thread [thread.thread.this]

```
namespace std {
  namespace this_thread {
    thread::id get_id();

    void yield();
    template <class Clock, class Duration>
      void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      void sleep_for(const chrono::duration<Rep, Period>& rel_time);
  }
}
```

`thread::id this_thread::get_id();`

1 *Returns:* An object of type `thread::id` that uniquely identifies the current thread of execution. No other thread of execution shall have this id and this thread of execution shall always have this id. The object returned shall not compare equal to a default constructed `thread::id`.

2 *Throws:* Nothing.

`void this_thread::yield();`

3 *Effects:* Offers the operating system the opportunity to schedule another thread.

4 *Synchronization:* None.

5 *Throws:* Nothing.

```
template <class Clock, class Duration>
  void sleep_until(const chrono::time_point<Clock, Duration>& abs_time);
```

6 *Effects:* Blocks the calling thread at least until the time specified by `abs_time`.

7 *Synchronization:* None.

8 *Throws:* Nothing.

```
template <class Rep, class Period>
  void sleep_for(const chrono::duration<Rep, Period>& rel_time);
```

9 *Effects:* Blocks the calling thread for at least the time specified by `rel_time`.

10 *Synchronization:* None.

11 *Throws:* Nothing.

30.3 Mutual exclusion [thread.mutex]

1 This section provides mechanisms for mutual exclusion: mutexes, locks, and call once. These mechanisms ease the production of race-free programs (1.10).

Header <mutex> synopsis

```
namespace std {
  class mutex;
  class recursive_mutex;
  class timed_mutex;
  class recursive_timed_mutex;
```

```
  struct defer_lock_t;
  struct try_to_lock_t;
  struct adopt_lock_t;

  extern const defer_lock_t  defer_lock;
  extern const try_to_lock_t try_to_lock;
  extern const adopt_lock_t  adopt_lock;

  template <class Mutex> class lock_guard;
  template <class Mutex> class unique_lock;

  template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y);
  template <class Mutex>
    void swap(unique_lock<Mutex>&& x, unique_lock<Mutex>& y);
  template <class Mutex>
    void swap(unique_lock<Mutex>& x, unique_lock<Mutex>&& y);

  template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
  template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);

  struct once_flag {
    constexpr once_flag();

    once_flag(const once_flag&) = delete;
    once_flag& operator=(const once_flag&) = delete;
  };

  template<class Callable, class ...Args>
    void call_once(once_flag& flag, Callable func, Args&&... args);
}
```

30.3.1 `Mutex` requirements [thread.mutex.requirements]

1 A mutex object facilitates protection against data races and allows thread-safe synchronization of data between threads. A thread *owns* a mutex from the time it successfully calls one of the lock functions until it calls unlock. Mutexes may be either recursive or non-recursive, and may grant simultaneous ownership to one or many threads. The mutex types supplied by the standard library provide exclusive ownership semantics: only one thread may own the mutex at a time. Both recursive and non-recursive mutexes are supplied.

2 This section describes requirements on template argument types used to instantiate templates defined in the C++ standard library. The template definitions in the C++ standard library refer to the named `Mutex` requirements whose details are set out below. In this description, `m` is an object of a `Mutex` type.

3 A `Mutex` type shall be `DefaultConstructible` and `Destructible`. If initialization of an object of a `Mutex` type fails, an exception of type `std::system_error` shall be thrown. A `Mutex` type shall not be copyable nor movable.

4 *Error conditions:*

— `not_enough_memory` — if there is not enough memory to construct the mutex object.

— `resource_unavailable_try_again` — if any native handle type manipulated is not available.

— `operation_not_permitted` — if the thread does not have the necessary permission to change the state of the mutex object.

— `device_or_resource_busy` — if any native handle type manipulated is already locked.

— `invalid_argument` — if any native handle type manipulated as part of mutex construction is incorrect.

5 The implementation shall provide lock and unlock operations, as described below. The implementation shall serialize those operations. [*Note:* Construction and destruction of an object of a `Mutex` type need not be thread-safe; other synchronization should be used to ensure that `Mutex` objects are initialized and visible to other threads. — *end note*]

6 The expression `m.lock()` shall be well-formed and have the following semantics:

7 *Effects:* Blocks the calling thread until ownership of the mutex can be obtained for the calling thread.

8 *Postcondition:* The calling thread owns the mutex.

9 *Return type:* `void`

10 *Synchronization:* Prior `unlock()` operations on the same object shall *synchronize with* (1.10) this operation.

11 *Throws:* `std::system_error` when the effects or postcondition cannot be achieved.

12 *Error conditions:*

— `operation_not_permitted` — if the thread does not have the necessary permission to change the state of the mutex.

— `resource_deadlock_would_occur` — if the current thread already owns the mutex and is able to detect it.

— `device_or_resource_busy` — if the mutex is already locked and blocking is not possible.

13 The expression `m.try_lock()` shall be well-formed and have the following semantics:

14 *Effects:* Attempts to obtain ownership of the mutex for the calling thread without blocking. If ownership is not obtained, there is no effect and `try_lock()` immediately returns. An implementation may fail to obtain the lock even if it is not held by any other thread. [*Note:* This spurious failure is normally uncommon, but allows interesting implementations based on a simple `compare_exchange` (29). — *end note*]

15 *Return type:* `bool`

16 *Returns:* `true` if ownership of the mutex was obtained for the calling thread, otherwise `false`.

17 *Synchronization:* If `try_lock()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.10) this operation. [*Note:* Since `lock()` does not synchronize with a failed subsequent `try_lock()`, the visibility rules are weak enough that little would be known about the state after a failure, even in the absence of spurious failures. — *end note*]

18 *Throws:* Nothing.

19 The expression `m.unlock()` shall be well-formed and have the following semantics:

20 *Precondition:* The calling thread shall own the mutex.

21 *Effects:* Releases the calling thread's ownership of the mutex.

22 *Return type:* `void`

23 *Synchronization:* This operation *synchronizes with* (1.10) subsequent lock operations that obtain ownership on the same object.

24 *Throws:* Nothing.

30.3.1.1 Class `mutex` [thread.mutex.class]

```
namespace std {
  class mutex {
  public:
    mutex();
    ~mutex();

    mutex(const mutex&) = delete;
    mutex& operator=(const mutex&) = delete;

    void lock();
    bool try_lock();
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.1.3
    native_handle_type native_handle();                // See 30.1.3
  };
}
```

1 The class `mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a mutex object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the owning thread has released ownership with a call to `unlock()`.

2 The class `mutex` shall satisfy all the `Mutex` requirements (30.3.1). It shall be a standard-layout class (9).

3 The behavior of a program is undefined if:

— it destroys a `mutex` object owned by any thread,

— a thread that owns a `mutex` object calls `lock()` or `try_lock()` on that object, or

— a thread terminates while owning a `mutex` object.

30.3.1.2 Class `recursive_mutex` [thread.mutex.recursive]

```
namespace std {
  class recursive_mutex {
  public:
    recursive_mutex();
    ~recursive_mutex();

    recursive_mutex(const recursive_mutex&) = delete;
    recursive_mutex& operator=(const recursive_mutex&) = delete;

    void lock();
    bool try_lock();
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.1.3
    native_handle_type native_handle();                // See 30.1.3
```

```
  };
}
```

1 The class `recursive_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`) until the first thread has completely released ownership.

2 The class `recursive_mutex` shall satisfy all the Mutex requirements (30.3.1). It shall be a standard-layout class (9).

3 A thread that owns a `recursive_mutex` object may acquire additional levels of ownership by calling `lock()` or `try_lock()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_mutex` object, additional calls to `try_lock()` shall fail, and additional calls to `lock()` shall throw an exception of type `std::system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()` and `try_lock()`. Only when all levels of ownership have been released may ownership be acquired by another thread.

4 The behavior of a program is undefined if:

— it destroys a `recursive_mutex` object owned by any thread or

— a thread terminates while owning a `recursive_mutex` object.

30.3.2 `TimedMutex` requirements [thread.timedmutex.requirements]

1 A `TimedMutex` type shall meet the requirements for a `Mutex` type. In addition, it shall meet the requirements set out in this Clause 30.3.2, where `rel_time` denotes an instantiation of `duration` (20.8.3) and `abs_time` denotes an instantiation of `time_point` (20.8.4).

2 The expression `m.try_lock_for(rel_time)` shall be well-formed and have the following semantics:

3 *Precondition:* If the tick `period` of `rel_time` is not exactly convertible to the native tick `period`, the `duration` shall be rounded up to the nearest native tick `period`.

4 *Effects:* The function attempts to obtain ownership of the mutex within the time specified by `rel_time`. If the time specified by `rel_time` is less than or equal to 0, the function attempts to obtain ownership without blocking (as if by calling `try_lock()`). The function shall return within the time specified by `rel_time` only if it has obtained ownership of the mutex object. [*Note:* As with `try_lock()`, there is no guarantee that ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — *end note*]

5 *Return type:* `bool`

6 *Returns:* `true` if ownership was obtained, otherwise `false`.

7 *Synchronization:* If `try_lock_for()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.10) this operation.

8 *Throws:* Nothing.

9 The expression `m.try_lock_until(abs_time)` shall be well-formed and have the following semantics:

10 *Effects:* The function attempts to obtain ownership of the mutex by the time specified by `abs_time`. If `abs_time` has already passed, the function attempts to obtain ownership without blocking (as if by calling `try_lock()`). The function shall return before the time specified by `abs_time` only if it has obtained ownership of the mutex object. [*Note:* As with `try_lock()`, there is no guarantee that

ownership will be obtained if the lock is available, but implementations are expected to make a strong effort to do so. — *end note*]

11 *Return type:* `bool`

12 *Returns:* `true` if ownership was obtained, otherwise `false`.

13 *Synchronization:* If `try_lock_until()` returns `true`, prior `unlock()` operations on the same object *synchronize with* (1.10) this operation.

14 *Throws:* Nothing.

30.3.2.1 Class `timed_mutex` [thread.timedmutex.class]

```
namespace std {
  class timed_mutex {
  public:
    timed_mutex();
    ~timed_mutex();

    timed_mutex(const timed_mutex&) = delete;
    timed_mutex& operator=(const timed_mutex&) = delete;

    void lock();
    bool try_lock();
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.1.3
    native_handle_type native_handle();                // See 30.1.3
  };
}
```

1 The class `timed_mutex` provides a non-recursive mutex with exclusive ownership semantics. If one thread owns a `timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has released ownership with a call to `unlock()` or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

2 The class `timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.3.2). It shall be a standard-layout class (9).

3 The behavior of a program is undefined if:

— it destroys a `timed_mutex` object owned by any thread,

— a thread that owns a `timed_mutex` object calls `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object, or

— a thread terminates while owning a `timed_mutex` object.

30.3.2.2 Class `recursive_timed_mutex` [thread.timedmutex.recursive]

```
namespace std {
  class recursive_timed_mutex {
  public:
    recursive_timed_mutex();
    ~recursive_timed_mutex();

    recursive_timed_mutex(const recursive_timed_mutex&) = delete;
    recursive_timed_mutex& operator=(const recursive_timed_mutex&) = delete;

    void lock();
    bool try_lock();
    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
    void unlock();

    typedef implementation-defined native_handle_type; // See 30.1.3
    native_handle_type native_handle();                // See 30.1.3
  };
}
```

1 The class `recursive_timed_mutex` provides a recursive mutex with exclusive ownership semantics. If one thread owns a `recursive_timed_mutex` object, attempts by another thread to acquire ownership of that object will fail (for `try_lock()`) or block (for `lock()`, `try_lock_for()`, and `try_lock_until()`) until the owning thread has completely released ownership or the call to `try_lock_for()` or `try_lock_until()` times out (having failed to obtain ownership).

2 The class `recursive_timed_mutex` shall satisfy all of the `TimedMutex` requirements (30.3.2). It shall be a standard-layout class (9).

3 A thread that owns a `recursive_timed_mutex` object may acquire additional levels of ownership by calling `lock()`, `try_lock()`, `try_lock_for()`, or `try_lock_until()` on that object. It is unspecified how many levels of ownership may be acquired by a single thread. If a thread has already acquired the maximum level of ownership for a `recursive_timed_mutex` object, additional calls to `try_lock()`, `try_lock_for()`, or `try_lock_until()` shall fail, and additional calls to `lock()` shall throw an exception of type `std::system_error`. A thread shall call `unlock()` once for each level of ownership acquired by calls to `lock()`, `try_lock()`, `try_lock_for()`, and `try_lock_until()`. Only when all levels of ownership have been released may ownership of the object be acquired by another thread.

4 The behavior of a program is undefined if:

— it destroys a `recursive_timed_mutex` object owned by any thread, or

— a thread terminates while owning a `recursive_timed_mutex` object.

30.3.3 Locks [thread.lock]

1 A *lock* is an object that holds a reference to a mutex and may unlock the mutex during the lock's destruction (such as when leaving block scope). A thread of execution may use a lock to aid in managing mutex ownership in an exception safe manner. A lock is said to *own* a mutex if it is currently managing the ownership of that mutex for a thread of execution. A lock does not manage the lifetime of the mutex it references. [*Note:* Locks are intended to ease the burden of unlocking the mutex under both normal and exceptional circumstances. — *end note*]

2 Some lock constructors take tag types which describe what should be done with the mutex object during the lock's constuction.

```
namespace std {
  struct defer_lock_t  { };      // do not acquire ownership of the mutex
  struct try_to_lock_t { };      // try to acquire ownership of the mutex
                                 // without blocking
  struct adopt_lock_t  { };      // assume the calling thread has already
                                 // obtained mutex ownership and manage it

  extern const defer_lock_t   defer_lock;
  extern const try_to_lock_t  try_to_lock;
  extern const adopt_lock_t   adopt_lock;
}
```

30.3.3.1 Class template `lock_guard` [thread.lock.guard]

```
namespace std {
  template <class Mutex>
  class lock_guard {
  public:
    typedef Mutex mutex_type;

    explicit lock_guard(mutex_type& m);
    lock_guard(mutex_type& m, adopt_lock_t);
    ~lock_guard();

    lock_guard(lock_guard const&) = delete;
    lock_guard& operator=(lock_guard const&) = delete;

  private:
    // exposition only:
    mutex_type& pm;
  };
}
```

1 An object of type `lock_guard` controls the ownership of a mutex object within a scope. A `lock_guard` object maintains ownership of a mutex object throughout the `lock_guard` object's lifetime. The behavior of a program is undefined if the mutex referenced by `pm` does not exist for the entire lifetime (3.8) of the `lock_guard` object.

```
explicit lock_guard(mutex_type& m);
```

2 *Precondition:* If `mutex_type` is not a recursive mutex, the calling thread does not own the mutex `m`.

3 *Effects:* `m.lock()`

4 *Postcondition:* `&pm == &m`

```
lock_guard(mutex_type& m, adopt_lock_t);
```

5 *Precondition:* The calling thread owns the mutex `m`.

6 *Postcondition:* `&pm == &m`

7 *Throws:* Nothing.

```
~lock_guard();
```

Effects: `pm.unlock()`

Throws: Nothing.

30.3.3.2 Class template `unique_lock` [thread.lock.unique]

```
namespace std {
  template <class Mutex>
  class unique_lock {
  public:
    typedef Mutex mutex_type;

    // 30.3.3.2.1 construct/copy/destroy
    unique_lock();
    explicit unique_lock(mutex_type& m);
    unique_lock(mutex_type& m, defer_lock_t);
    unique_lock(mutex_type& m, try_to_lock_t);
    unique_lock(mutex_type& m, adopt_lock_t);
    template <class Clock, class Duration>
      unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
    template <class Rep, class Period>
      unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
    ~unique_lock();

    unique_lock(unique_lock const&) = delete;
    unique_lock& operator=(unique_lock const&) = delete;

    unique_lock(unique_lock&& u);
    unique_lock& operator=(unique_lock&& u);

    // 30.3.3.2.2 locking
    void lock();
    bool try_lock();

    template <class Rep, class Period>
      bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
    template <class Clock, class Duration>
      bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);

    void unlock();

    // 30.3.3.2.3 modifiers
    void swap(unique_lock&& u);
    mutex_type *release();

    // 30.3.3.2.4 observers
    bool owns_lock() const;
    explicit operator bool () const;
    mutex_type* mutex() const;

  private:
    // exposition only:
    mutex_type *pm;
    bool owns;
  };
```

```
    template <class Mutex>
      void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y);
    template <class Mutex>
      void swap(unique_lock<Mutex>&& x, unique_lock<Mutex>& y);
    template <class Mutex>
      void swap(unique_lock<Mutex>& x, unique_lock<Mutex>&& y);
  }
```

1 An object of type `unique_lock` controls the ownership of a mutex within a scope. Mutex ownership may be acquired at construction or after construction, and may be transferred, after acquisition, to another `unique_lock` object. Objects of type `unique_lock` are not copyable but are movable. The behavior of a program is undefined if the contained pointer `pm` is not null and the mutex pointed to by `pm` does not exist for the entire remaining lifetime (3.8) of the `unique_lock` object.

30.3.3.2.1 `unique_lock` constructors, destructor, and assignment [thread.lock.unique.cons]

`unique_lock();`

1 *Effects:* Constructs an object of type `unique_lock`.

2 *Postconditions:* `pm == 0` and `owns == false`.

3 *Throws:* Nothing.

`explicit unique_lock(mutex_type& m);`

4 *Precondition:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

5 *Effects:* Constructs an object of type `unique_lock` and calls `m.lock()`.

6 *Postconditions:* `pm == &m` and `owns == true`.

`unique_lock(mutex_type& m, defer_lock_t);`

7 *Precondition:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

8 *Effects:* Constructs an object of type `unique_lock`.

9 *Postconditions:* `pm == &m` and `owns == false`.

10 *Throws:* Nothing.

`unique_lock(mutex_type& m, try_to_lock_t);`

11 *Precondition:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

12 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock()`.

13 *Postconditions:* `pm == &m` and `owns == res`, where `res` is the value returned by the call to `m.try_lock()`.

14 *Throws:* Nothing.

`unique_lock(mutex_type& m, adopt_lock_t);`

15 *Precondition:* The calling thread own the mutex.

16 *Effects:* Constructs an object of type `unique_lock`.

17 *Postconditions:* `pm == &m` and `owns == true`.

18 *Throws:* Nothing.

```
template <class Clock, class Duration>
  unique_lock(mutex_type& m, const chrono::time_point<Clock, Duration>& abs_time);
```

19 *Precondition:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

20 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock_until(abs_time)`.

21 *Postconditions:* `pm == &m` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_until(abs_time)`.

22 *Throws:* Nothing.

```
template <class Rep, class Period>
  unique_lock(mutex_type& m, const chrono::duration<Rep, Period>& rel_time);
```

23 *Precondition:* If `mutex_type` is not a recursive mutex the calling thread does not own the mutex.

24 *Effects:* Constructs an object of type `unique_lock` and calls `m.try_lock_for(rel_time)`.

25 *Postconditions:* `pm == &m` and `owns == res`, where `res` is the value returned by the call to `m.try_lock_for(rel_time)`.

26 *Throws:* Nothing.

```
unique_lock(unique_lock&& u);
```

27 *Postconditions:* `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == 0` and `u.owns == false`.

28 *Throws:* Nothing.

```
unique_lock& operator=(unique_lock&& u);
```

29 *Effects:* If `owns` calls `pm->unlock()`.

30 *Postconditions:* `pm == u_p.pm` and `owns == u_p.owns` (where `u_p` is the state of `u` just prior to this construction), `u.pm == 0` and `u.owns == false`.

31 *Throws:* Nothing.

32 [*Note:* With a recursive mutex it is possible for both `*this` and `u` to own the same mutex before the assignment. In this case, `*this` will own the mutex after the assignment and `u` will not. — *end note*]

```
~unique_lock();
```

33 *Effects:* If `owns` calls `pm->unlock()`.

34 *Throws:* Nothing.

30.3.3.2.2 `unique_lock` locking [thread.lock.unique.locking]

```
void lock();
```

1 *Effects:* `pm->lock()`

2 *Postcondition:* `owns == true`

3 *Throws:* std::system_error when the postcondition cannot be achieved.

4 *Error conditions:*

— `operation_not_permitted` — if `pm` is null.

— `resource_deadlock_would_occur` — if the current thread already owns the mutex (i.e. on entry, `owns` is true).

```
bool try_lock();
```

Effects: `pm->try_lock()`

Returns: The value returned by the call to `try_lock()`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock()`.

Throws: std::system_error when the postcondition cannot be achieved.

Error conditions:

— `operation_not_permitted` — if `pm` is null.

— `resource_deadlock_would_occur` — if the current thread already owns the mutex (i.e. on entry, `owns` is true).

```
template <class Clock, class Duration>
  bool try_lock_until(const chrono::time_point<Clock, Duration>& abs_time);
```

Effects: `pm->try_lock_until(abs_time)`

Returns: The value returned by the call to `try_lock_until(abs_time)`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock_until(abs_time)`.

Throws: std::system_error when the postcondition cannot be achieved.

Error conditions:

— `operation_not_permitted` — if `pm` is null.

— `resource_deadlock_would_occur` — if the current thread already owns the mutex (i.e. on entry, `owns` is true).

```
template <class Rep, class Period>
  bool try_lock_for(const chrono::duration<Rep, Period>& rel_time);
```

Effects: `pm->try_lock_for(rel_time)`.

Returns: The value returned by the call to `try_lock_until(rel_time)`.

Postcondition: `owns == res`, where `res` is the value returned by the call to `try_lock_for(rel_time)`.

Throws: std::system_error when the postcondition cannot be achieved.

Error conditions:

— `operation_not_permitted` — if `pm` is null.

— `resource_deadlock_would_occur` — if the current thread already owns the mutex (i.e. on entry, `owns` is true).

```
void unlock();
```

Effects: `pm->unlock()`

Postcondition: `owns == false`

22 *Throws:* std::system_error when the postcondition cannot be achieved.

23 *Error conditions:*

— `operation_not_permitted` — if on entry `owns` is false.

30.3.3.2.3 `unique_lock` modifiers [thread.lock.unique.mod]

```
void swap(unique_lock&& u);
```

1 *Effects:* Swaps the data members of `*this` and `u`.

2 *Throws:* Nothing.

```
mutex_type *release();
```

3 *Returns:* The previous value of `pm`.

4 *Postconditions:* `pm == 0` and `owns == false`.

5 *Throws:* Nothing.

```
template <class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>& y);
template <class Mutex>
  void swap(unique_lock<Mutex>&& x, unique_lock<Mutex>& y);
template <class Mutex>
  void swap(unique_lock<Mutex>& x, unique_lock<Mutex>&& y);
```

6 *Effects:* `x.swap(y)`

7 *Throws:* Nothing.

30.3.3.2.4 `unique_lock` observers [thread.lock.unique.obs]

```
bool owns_lock() const;
```

1 *Returns:* `owns`

2 *Throws:* Nothing.

```
explicit operator bool() const;
```

3 *Returns:* `owns`

4 *Throws:* Nothing.

```
mutex_type *mutex() const;
```

5 *Returns:* `pm`

6 *Throws:* Nothing.

30.3.4 Generic locking algorithms [thread.lock.algorithm]

```
template <class L1, class L2, class... L3> int try_lock(L1&, L2&, L3&...);
```

1 *Requires:* Each template parameter type shall meet the `Mutex` requirements, except that a call to `try_lock()` may throw an exception. [*Note:* The `unique_lock` class template meets these requirements when suitably instantiated. — *end note*]

2 *Effects:* Calls `try_lock()` for each argument in order beginning with the first until all arguments have been processed or a call to `try_lock()` fails, either by returning `false` or by throwing an exception. If a call to `try_lock()` fails, `unlock()` shall be called for all prior arguments.

3 *Returns:* `-1` if all calls to `try_lock()` returned `true`, otherwise a 0-based index value that indicates the argument for which `try_lock()` returned `false`. [*Note:* On return, either all arguments will be locked or none will be locked. — *end note*]

```
template <class L1, class L2, class... L3> void lock(L1&, L2&, L3&...);
```

4 *Requires:* Each template parameter type shall meet the `Mutex` requirements, except that a call to `try_lock()` may throw an exception. [*Note:* The `unique_lock` class template meets these requirements when suitably instantiated. — *end note*]

5 *Effects:* All arguments are locked via a sequence of calls to `lock()`, `try_lock()`, or `unlock()` on each argument. The sequence of calls shall not result in deadlock, but is otherwise unspecified. [*Note:* A deadlock avoidance algorithm such as try-and-back-off must be used, but the specific algorithm is not specified to avoid over-constraining implementations. — *end note*] If a call to `lock()` or `try_lock()` throws an exception, `unlock()` shall be called for any argument that had been locked by a call to `lock()` or `try_lock()`.

30.3.5 Call once [thread.once]

The class `once_flag` is an opaque data structure that `call_once` uses to initialize data without causing a data race or deadlock.

30.3.5.1 Struct `once_flag` [thread.once.onceflag]

```
constexpr once_flag();
```

1 *Effects:* Constructs an object of type `once_flag`.

2 *Synchronization:* The construction of a `once_flag` object is not synchronized.

3 *Postcondition:* The object's internal state is set to indicate to an invocation of `call_once` with the object as its initial argument that no function has been called.

4 *Throws:* nothing.

30.3.5.2 Function `call_once` [thread.once.callonce]

```
template<class Callable, class ...Args>
  void call_once(once_flag& flag, Callable func, Args&&... args);
```

1 *Requires:* The template parameters `Callable` and each `Ti` in `Args` shall be `CopyConstructible` if an lvalue and otherwise `MoveConstructible`. *`INVOKE`*`(func, w1, w2, ..., wN)` (20.6.2) shall be a valid expression for some values *w1, w2, ..., wN*, where `N == sizeof...(Args)`.

2 *Effects:* Calls to `call_once` on the same `once_flag` object are serialized. If there has been a prior effective call to `call_once` on the same `once_flag` object, the call to `call_once` returns without invoking `func`. If there has been no prior effective call to `call_once` on the same `once_flag` object, the argument `func` (or a copy thereof) is called as if by invoking `func(args)`. The call to `call_once` is effective if and only if `func(args)` returns without throwing an exception. If an exception is thrown it is propagated to the caller.

3 *Synchronization:* The completion of an effective call to `call_once` on a `once_flag` object *synchronizes with* (1.10) all subsequent calls to `call_once` on the same `once_flag` object.

4 *Throws:* `std::system_error` when the effects cannot be achieved, or any exception thrown by `func`.

5 *Error conditions:*

— `invalid_argument` — if the `once_flag` object is no longer valid.

[*Example:*

```
// global flag, regular function
void init();
std::once_flag flag;

void f() {
  std::call_once(flag,init);
}

// function static flag, function object
struct initializer {
  void operator()();
};

void g() {
  static std::once_flag flag2;
  std::call_once(flag2,initializer);
}

// object flag, member function
class information {
  std::once_flag verified;
  void verifier();
public:
  void verify() { std::call_once(verified,verifier); }
};
```

— *end example*]

30.4 Condition variables [thread.condition]

1 Condition variables provide synchronization primitives used to block a thread until notified by some other thread that some condition is met or until a system time is reached. Class `condition_variable` provides a condition variable that can only wait on a `Lock`, allowing maximum efficiency on some platforms. Class `condition_variable_any` provides a general condition variable that can wait on user-supplied lock types.

2 Condition variables permit concurrent invocation of the `wait`, `wait_for`, `wait_until`, `notify_one` and `notify_all` member functions.

3 The execution of `notify_one` and `notify_all` shall be atomic. The execution of `wait`, `wait_for`, and `wait_until` shall be performed in three atomic parts:

1. the release of the mutex, and entry into the waiting state;
2. the unblocking of the wait; and
3. the reacquisition of the lock.

4 The implementation shall behave as if `notify_one`, `notify_all`, and each part of the `wait`, `wait_for`, and `wait_until` executions are executed in some unspecified total order.

5 Condition variable construction and destruction need not be synchronized.

Header `condition_variable` synopsis

```
namespace std {
  class condition_variable;
  class condition_variable_any;
}
```

30.4.1 Class `condition_variable` [thread.condition.condvar]

```
namespace std {
  class condition_variable {
  public:

    condition_variable();
    ~condition_variable();

    condition_variable(const condition_variable&) = delete;
    condition_variable& operator=(const condition_variable&) = delete;

    void notify_one();
    void notify_all();
    void wait(unique_lock<mutex>& lock);
    template <class Predicate>
      void wait(unique_lock<mutex>& lock, Predicate pred);
    template <class Clock, class Duration>
      bool wait_until(unique_lock<mutex>& lock,
                      const chrono::time_point<Clock, Duration>& abs_time);
    template <class Clock, class Duration, class Predicate>
      bool wait_until(unique_lock<mutex>& lock,
                      const chrono::time_point<Clock, Duration>& abs_time,
                      Predicate pred);

    template <class Rep, class Period>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time);
    template <class Rep, class Period, class Predicate>
      bool wait_for(unique_lock<mutex>& lock,
                    const chrono::duration<Rep, Period>& rel_time,
                    Predicate pred);

    typedef implementation-defined native_handle_type; // See 30.1.3
    native_handle_type native_handle();                // See 30.1.3
  };
}
```

1 The class `condition_variable` shall be a standard-layout class (9).

`condition_variable();`

2 *Effects:* Constructs an object of type `condition_variable`.

3 *Error conditions:*

— `not_enough_memory` — if a memory limitation prevents initialization.

— `resource_unavailable_try_again` — if some non-memory resource limitation prevents initialization.

— `device_or_resource_busy` — if attempting to initialize a previously-initialized but as of yet undestroyed `condition_variable`.

```
~condition_variable();
```

4 *Precondition:* There shall be no thread blocked on `*this`. [*Note:* That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. Beware that destroying a `condition_variable` object while the corresponding predicate is false is likely to lead to undefined behavior. — *end note*]

5 *Effects:* Destroys the object.

6 *Throws:* Nothing.

```
void notify_one();
```

7 *Effects:* If any threads are blocked waiting for `*this`, unblocks one of those theads.

```
void notify_all();
```

8 *Effects:* Unblocks all threads that are blocked waiting for `*this`.

```
void wait(unique_lock<mutex>& lock);
```

9 *Precondition:* `lock` is locked by the calling thread, and either

— no other thread is waiting on this `condition_variable` object or

— `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting threads (via `wait` or `timed_wait`).

10 *Effects:*

— Atomically calls `lock.unlock()` and blocks on `*this`.

— When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.

— The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, or spuriously.

— If the function exits via an exception, `lock.unlock()` shall be called prior to exiting the function scope.

11 *Postcondition:* `lock` is locked by the calling thread.

12 *Throws:* `std::system_error` when the effects or postcondition cannot be achieved.

13 *Error conditions:*

— equivalent error condition from `lock.lock()` or `lock.unlock()`.

```
template <class Predicate>
  void wait(unique_lock<mutex>& lock, Predicate pred);
```

14 *Effects:*

```
while (!pred())
  wait(lock);
```

```
template <class Clock, class Duration>
  bool wait_until(unique_lock<mutex>& lock,
                  const chrono::time_point<Clock, Duration>& abs_time);
```

15 *Precondition:* `lock` is locked by the calling thread, and either

- no other thread is waiting on this `condition_variable` object or
- `lock.mutex()` returns the same value for each of the `lock` arguments supplied by all concurrently waiting threads (via `wait` , `wait_for` or `wait_until`.).

16 *Effects:*

- Atomically calls `lock.unlock()` and blocks on `*this`.
- When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
- The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, by the current time exceeding `abs_time`, or spuriously.
- If the function exits via an exception, `lock.unlock()` shall be called prior to exiting the function scope.

17 *Postcondition:* `lock` is locked by the calling thread.

18 *Returns:* `Clock::now() < abs_time`.

19 *Throws:* `std::system_error` when the effects or postcondition cannot be achieved.

20 *Error conditions:*

- `operation_not_permitted` — if the thread does not own the lock.
- equivalent error condition from `lock.lock()` or `lock.unlock()`.

```
template <class Rep, class Period>
  bool wait_for(unique_lock<mutex>& lock,
                const chrono::duration<Rep, Period>& rel_time);
```

21 *Effects:*

```
wait_until(lock, chrono::monotonic_clock::now() + rel_time)
```

22 *Returns:* `false` if the call is returning because the time duration specified by `rel_time` has elapsed, otherwise `true`.

```
template <class Clock, class Duration, class Predicate>
  bool wait_until(unique_lock<mutex>& lock,
                  const chrono::time_point<Clock, Duration>& abs_time,
                  Predicate pred);
```

23 *Effects:*

```
while (!pred())
  if (!wait_until(lock, abs_time))
    return pred();
return true;
```

24 *Returns:* `pred()`

25 [*Note:* The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

```
template <class Rep, class Period, class Predicate>
  bool wait_for(unique_lock<mutex>& lock,
                const chrono::duration<Rep, Period>& rel_time,
                Predicate pred);
```

26 *Effects:*

```
wait_until(lock, chrono::monotonic_clock::now() + rel_time, std::move(pred))
```

27 [*Note:* There is no blocking if `pred()` is initially `true`, even if the timeout has already expired. — *end note*]

28 *Returns:* `pred()`

29 [*Note:* The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

30.4.2 Class `condition_variable_any` [thread.condition.condvarany]

1 A `Lock` type shall meet the requirements for a `Mutex` type, except that `try_lock` is not required. [*Note:* All of the standard mutex types meet this requirement. — *end note*]

```
namespace std {
  class condition_variable_any {
  public:
    condition_variable_any();
    ~condition_variable_any();

    condition_variable_any(const condition_variable_any&) = delete;
    condition_variable_any& operator=(const condition_variable_any&) = delete;

    void notify_one();
    void notify_all();
    template <class Lock>
      void wait(Lock& lock);
    template <class Lock, class Predicate>
      void wait(Lock& lock, Predicate pred);

    template <class Lock, class Clock, class Duration>
      bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
    template <class Lock, class Clock, class Duration, class Predicate>
      bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time, Predicate pred);
    template <class Lock, class Rep, class Period>
      bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
    template <class Lock, class Rep, class Period, class Predicate>
      bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);

    typedef implementation-defined native_handle_type; // See 30.1.3
    native_handle_type native_handle();                // See 30.1.3
  };
}
```

```
condition_variable_any();
```

2 *Effects:* Constructs an object of type `condition_variable_any`.

```
~condition_variable_any();
```

3 *Precondition:* There shall be no thread blocked on `*this`. [*Note:* That is, all threads shall have been notified; they may subsequently block on the lock specified in the wait. Beware that destroying a `condition_variable_any` object while the corresponding predicate is false is likely to lead to undefined behavior. — *end note*]

4 *Effects:* Destroys the object.

5 *Throws:* Nothing.

```
void notify_one();
```

6 *Effects:* If any threads are blocked waiting for `*this`, unblocks one of those theads.

```
void notify_all();
```

7 *Effects:* Unblocks all threads that are blocked waiting for `*this`.

```
template <class Lock>
  void wait(Lock& lock);
```

8 *Effects:*

- Atomically calls `lock.unlock()` and blocks on `*this`.
- When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
- The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, or spuriously.
- If the function exits via an exception, `lock.unlock()` shall be called prior to exiting the function scope.

9 *Postcondition:* `lock` is locked by the calling thread.

10 *Throws:* `std::system_error` when the effects or postcondition cannot be achieved.

11 *Error conditions:*

- equivalent error condition from `lock.lock()` or `lock.unlock()`.

```
template <class Lock, class Predicate>
  void wait(Lock& lock, Predicate pred);
```

12 *Effects:*

```
while (!pred())
  wait(lock);
```

```
template <class Lock, class Clock, class Duration>
  bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& abs_time);
```

13 *Effects:*

- Atomically calls `lock.unlock()` and blocks on `*this`.
- When unblocked, calls `lock.lock()` (possibly blocking on the lock) and returns.
- The function will unblock when signaled by a call to `notify_one()`, a call to `notify_all()`, by the current time exceeding `abs_time`, or spuriously.
- If the function exits via an exception, `lock.unlock()` shall be called prior to exiting the function scope.

14 *Postcondition:* `lock` is locked by the calling thread.

15 *Returns:* `Clock::now() < abs_time`.

16 *Throws:* `std::system_error` when the returned value, effects, or postcondition cannot be achieved.

17 *Error conditions:*

— equivalent error condition from `lock.lock()` or `lock.unlock()`.

```
template <class Lock, class Rep, class Period>
  bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time);
```

18 *Effects:*

```
wait_until(lock, chrono::monotonic_clock::now() + rel_time)
```

19 *Returns:* `false` if the call is returning because the time duration specified by `rel_time` has elapsed, otherwise `true`.

```
template <class Lock, class Duration, class Predicate>
  bool wait_until(Lock& lock, const chrono::time_point<Clock, Duration>& rel_time, Predicate pred);
```

20 *Effects:*

```
while (!pred())
  if (!wait_until(lock, abs_time))
    return pred();
return true;
```

21 *Returns:* `pred()`

22 [*Note:* The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

```
template <class Lock, class Rep, class Period, class Predicate>
  bool wait_for(Lock& lock, const chrono::duration<Rep, Period>& rel_time, Predicate pred);
```

23 *Effects:*

```
wait_until(lock, chrono::monotonic_clock::now() + rel_time, std::move(pred))
```

24 [*Note:* There is no blocking if `pred()` is initially `true`, even if the timeout has already expired. — *end note*]

25 *Returns:* `pred()`

26 [*Note:* The returned value indicates whether the predicate evaluates to `true` regardless of whether the timeout was triggered. — *end note*]

30.5 Futures [futures]

30.5.1 Overview [futures.overview]

1 30.5 describes components that a C++ program can use to retrieve in one thread the result (value or exception) from a function that has run in another thread. [*Note:* these components are not restricted to multi-threaded programs but can be useful in single-threaded programs as well. — *end note*]

Header `<future>` synopsis

```
namespace std {
  enum class future_errc {
    broken_promise,
    future_already_retrieved,
    promise_already_satisfied
  };

  concept_map ErrorCodeEnum<future_errc> { }

  constexpr error_code make_error_code(future_errc e);
  constexpr error_condition make_error_condition(future_errc e);

  extern const error_category* const future_category;

  class future_error;

  template <class R> class unique_future;
  template <class R> class unique_future<R&>;
  template <> class unique_future<void>;
  template <class R> class shared_future;
  template <class R> class shared_future<R&>;
  template <> class shared_future<void>;
  template <class R> class promise;
  template <class R> class promise<R&;
  template <> class promise<void>;

  template <class R, class Alloc>
    struct uses_allocator<promise<R>, Alloc>;
  template <class R>
    struct constructible_with_allocator_prefix<promise<R> >;

  template <class> class packaged_task;   // undefined
  template <class R, class... Argtypes>
    class packaged_task<R(Argtypes...)>;
}
```

30.5.2 Error handling [futures.errors]

```
extern const error_category* const future_category;
```

1 `future_category` shall point to a statically initialized object of a type derived from class `error_category`.

2 The object's `default_error_condition` and equivalent virtual functions shall behave as specified for the class error_category. The object's `name` virtual function shall return a pointer to the string `"future"`.

```
constexpr error_code make_error_code(future_errc e);
```

3 *Returns:* `error_code(static_cast<int>(e), future_category)`.

```
constexpr error_code make_error_condition(future_errc e);
```

4 *Returns:* `error_condition(static_cast<int>(e), future_category)`.

30.5.3 Class `future_error` [futures.future_error]

```
namespace std {
  class future_error : public logic_error {
  public:
    future_error(error_code ec);  // exposition only

    const error_code& code() const throw();
    const char*       what() const throw();
  };
}
```

```
const error_code& code() const throw();
```

1 *Returns:* the value of `ec` that was passed to the object's constructor.

```
const char *what() const throw();
```

2 *Returns:* an NTBS incorporating `code().message()`.

30.5.4 Class template `unique_future` [futures.unique_future]

```
namespace std {
  template <class R>
  class unique_future {
  public:
    unique_future(unique_future &&);
    unique_future(const unique_future& rhs) = delete;
    ~unique_future();
    unique_future& operator=(const unique_future& rhs) = delete;

    // retrieving the value
    see below get();

    // functions to check state and wait for ready
    bool is_ready() const;
    bool has_exception() const;
    bool has_value() const;

    void wait() const;
    template <class Rep, class Period>>
      bool wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
      bool wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
  };
}
```

1 The implementation shall provide the template `unique_future` and two specializations, `unique_future<R&>` and `unique_future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

```
unique_future(unique_future&& rhs);
```

2 *Effects:* move constructs a `unique_future` object whose associated state is the same as the state of `rhs` before. The associated state is the state and the (possibly not yet evaluated) result (value or exception) associated with the `promise` object that provided the original future.

3 *Postcondition:* `rhs` can be safely destroyed.

```
~unique_future();
```

4 *Effects:* destroys `*this` and its associated state if no other object refers to that.

```
R&& unique_future::get();
R& unique_future<R&>::get();
void unique_future<void>::get();
```

5 *Note:* as described above, the template and its two required specializations differ only in the return type and return value of the member function `get`.

6 *Synchronization:* if `*this` is associated with a `promise` object, the completion of `set_value()` or `set_exception()` to that `promise` happens before (1.10) `get()` returns.

7 *Returns:*

— `unique_future::get()` returns an rvalue-reference to the value stored in the asynchronous result.

— `unique_future<R&>::get()` returns the stored reference.

— `unique_future<void>::get()` returns nothing.

8 *Throws:* the stored exception, if an exception was stored and not retrieved before.

9 *Remark:* the effect of calling `get()` a second time on the same `unique_future` object is unspecified.

```
bool is_ready() const;
```

10 *Returns:* `true` only if the associated state holds a value or an exception ready for retrieval.

Remark: the return value is unspecified after a call to `get()`.

```
bool has_exception() const;
```

11 *Returns:* `true` only if `is_ready() == true` and the associated state contains an exception.

```
bool has_value() const;
```

12 *Returns:* `true` only if `is_ready() == true` and the associated state contains a value.

```
void wait() const;
```

13 *Effects:* blocks until `*this` is ready.

14 *Synchronization:* if `*this` is associated with a `promise` object, the completion of `set_value()` or `set_exception()` to that `promise` happens before (1.10) `wait()` returns.

15 *Postcondition:* `is_ready() == true`

```
template <class Rep, class period>
bool wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

16 *Effects:* blocks until `*this` is ready or until `rel_time` has elapsed.

17 *Returns:* `true` only if the function returns because `*this` is ready.

18 *Postcondition:* `is_ready()` equals the return value.

```
template <class Clock, class Duration>
bool wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

19 Same as `wait_for`, except that it blocks until `abs_time` is reached if the associated state is not ready.

30.5.5 Class template shared_future [future.shared_future]

```
namespace std {
  template <class R>
  class shared_future {
  public:
    shared_future(const shared_future& rhs);
    shared_future(unique_future<R>);
    ~shared_future();
    shared_future & operator=(const shared_future& rhs) = delete;

    // retrieving the value
    see below get() const;

    // functions to check state and wait for ready
    bool is_ready() const;
    bool has_exception() const;
    bool has_value() const;

    void wait() const;
    template <class Rep, class Period>>
      bool wait_for(const chrono::duration<Rep, Period>& rel_time) const;
    template <class Clock, class Duration>
      bool wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
  };
}
```

1 The implementation shall provide the template `shared_future` and two specializations, `shared_future<R&>` and `shared_future<void>`. These differ only in the return type and return value of the member function `get`, as set out in its description, below.

`shared_future(const shared_future& rhs);`

2 *Effects:* copy constructs a `shared_future` object whose associated state is the same as the state of `rhs` before. The associated state is the state and the (possibly not yet evaluated) result (value or exception) associated with the `promise` object that provided the original future.

`shared_future(const unique_future<R> rhs);`

3 *Effects:* move constructs a `shared_future` object whose associated state is the same as the state of `rhs` before.

4 *Postcondition:* `rhs` can be safely destroyed.

`~shared_future();`

5 *Effects:* destroys `*this` and its associated state if no other object refers to that.

```
const R& shared_future::get() const;
R& shared_future<R&>::get() const;
void shared_future<void>::get() const;
```

6 *Note:* as described above, the template and its two required specializations differ only in the return type and return value of the member function `get`.

7 *Synchronization:* if `*this` is associated with a `promise` object, the completion of `set_value()` or `set_exception()` to that `promise` happens before (1.10) `get()` returns.

8 *Returns:*

— `unique_future::get()` returns a const reference to the value stored in the asynchronous result.

— `unique_future<R&>::get()` returns the stored reference.

— `unique_future<void>::get()` returns nothing.

9 *Throws:* the stored exception, if an exception was stored and not retrieved before.

```
bool is_ready() const;
```

10 *Returns:* `true` only if the associated state holds a value or an exception ready for retrieval.

```
bool has_exception() const;
```

11 *Returns:* `true` only if `is_ready() == true` and the associated state contains an exception.

```
bool has_value() const;
```

12 *Returns:* `true` only if `is_ready() == true` and the associated state contains a value.

```
void wait() const;
```

13 *Effects:* blocks until `*this` is ready.

14 *Synchronization:* if `*this` is associated with a `promise` object, the completion of `set_value()` or `set_exception()` to that `promise` happens before (1.10) `wait()` returns.

15 *Postcondition:* `is_ready() == true`

```
template <class Rep, class period>
bool wait_for(const chrono::duration<Rep, Period>& rel_time) const;
```

16 *Effects:* blocks until `*this` is ready or until `rel_time` has elapsed.

17 *Returns:* `true` only if the function returns because `*this` is ready.

18 *Postcondition:* `is_ready()` equals the return value.

```
template <class Clock, class Duration>
bool wait_until(const chrono::time_point<Clock, Duration>& abs_time) const;
```

19 Same as `wait_for`, except that it blocks until `abs_time` is reached if the associated state is not ready.

30.5.6 Class template `promise` [futures.promise]

```
namespace std {
  template <class R>
  class promise {
  public:
    promise();
    template <class Allocator>
      promise(allocator_arg_t, const Allocator& a);
    promise(promise&& rhs);
    template <class Allocator>
      promise(allocator_arg_t, const Allocator& a,
              promise& rhs);
    promise(const promise& rhs) = delete;
    ~promise();
```

```
    // assignment
    promise & operator=(promise&& rhs);
    promise & operator=(const promise& rhs) = delete;
    void swap(promise& other);

    // retrieving the result
    unique_future<R> get_future();

    // setting the result
    void set_value(const R& r);
    void set_value(see below);
    void set_exception(exception_ptr p);
  };
}
```

1 The implementation shall provide the template `promise` and two specializations, `promise<R&>` and `promise<void>`. These differ only in the argument type of the member function `set_value`, as set out in its description, below.

```
promise();
template <class Allocator>
  promise(allocator_arg_t, const Allocator& a);
```

2 *Effects:* constructs a `promise` object and an associated state. The second constructor uses the allocator `a` to allocate memory for the associated state.

```
promise(promise&& rhs);
template <class Allocator>
  promise(allocator_arg_t, const Allocator& a, promise& rhs);
```

3 *Effects:* move constructs a `promise` object whose associated state is the same as the state of `rhs` before.

4 *Postcondition:* `rhs` has no associated state.

```
~promise();
```

5 *Effects:* destroys `*this` and its associated state if no other object refers to it. If another object refers to the associated state of `*this` and that state is not ready, sets that state to ready and stores a `future_error` exception with error code `broken_promise` as result.

```
promise& operator=(promise&& rhs);
```

6 *Effects:* move assigns its associated state to `rhs`.

7 *Postcondition:* `*this` has no associated state.

8 *Returns:* `*this`.

9 *Throws:* nothing.

```
void swap(promise& other);
```

10 *Effects:* `swap(*this, other)`

```
unique_future<R> get_future();
```

11 *Returns:* a `unique_future<R>` object with the same associated state as `*this`.

12 *Throws:* `future_error` if `*this` has no associated state.

13 *Error conditions:* `future_already_retrieved` if `*this` has no associated state.

```
void set_value(const R& r);
void promise::set_value(R&& r);
void promise<R&>::set_value(r& r);
void promise<void>::set_value();
```

14 *Effects:* stores `r` in the associated state and sets that state to ready. Any blocking waits on the associated state are woken up.

15 *Throws:* `future_error` if its associated state is already ready.

16 *Error conditions:* `promise_already_satisfied` if its associated state is already ready.

```
void set_exception(exception_ptr p);
```

17 *Effects:* stores `p` in the associated state and sets that state to ready. Any blocking waits on the associated state are woken up.

18 *Throws:* `future_error` if its associated state is already ready.

19 *Error conditions:* `promise_already_satisfied` if its associated state is already ready.

30.5.7 Allocator templates [futures.allocators]

```
template <class R, class Alloc>
  struct uses_allocator<promise<R>, Alloc> : true_type { };
```

1 *Requires:* `Alloc` shall be an `Allocator` (20.7.2.2).

2 *Notes:* specialization of this trait informs other library components that `promise` can be constructed with an allocator, even though it does not have an `allocator_type` associated type.

```
template <class R>
  struct constructible_with_allocator_prefix<promise<R> > : true_type { };
```

3 *Notes:* specialization of this trait informs other library components that a `promise` can always be constructed with an allocator prefix argument.

30.5.8 Class template packaged_task [futures.task]

```
namespace std {
  template<class> class packaged_task; // undefined

  template<class R, class... ArgTypes>
  class packaged_task<R(ArgTypes...)> {
  public:
    typedef R result_type;

    // construction and destruction
    packaged_task();
    template <class F>
      explicit packaged_task(F f);
    template <class F, class Allocator>
      explicit packaged_task(allocator_arg_t, const Allocator& a, F f);
    explicit packaged_task(R(*f)());
    template <class F>
      explicit packaged_task(F&& f);
```

```
    template <class F, class Allocator>
      explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
    ~packaged_task();

    // no copy
    packaged_task(packaged_task&) = delete;
    packaged_task& operator=(packaged_task&) = delete;

    // move support
    packaged_task(packaged_task&& other);
    packaged_task& operator=(packaged_task&& other);
    void swap(packaged_task&& other);

    explicit operator bool() const;

    // result retrieval
    unique_future<R> get_future();

    // execution
    void operator()(ArgTypes... );

    void reset();
  };
}
```

`packaged_task();`

1 *Effects:* constructs a `packaged_task` object with no associated task.

2 *Throws:* nothing.

```
template <class F>
  packaged_task(F f);
template <class F, class Allocator>
  explicit packaged_task(allocator_arg_t, const Allocator& a, F f);
packaged_task(R(*f)());
template <class F>
  packaged_task(F&& f);
template <class F, class Allocator>
  explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
```

3 *Preconditions:* `f()` shall be a valid expression with a return type convertible to `R`. Invoking a copy of `f` shall behave the same as invoking `f`.

4 *Effects:* constructs a new `packaged_task` object with a copy of `f` stored as the object's associated task.

The constructors that take an `Allocator` argument use it to allocate memory needed to store the internal data structures.

5 *Throws:* any exceptions thrown by the copy or move constructor of `f`, or `std::bad_alloc` if memory for the internal data structures could not be allocated.

`packaged_task(packaged_task&& other);`

6 *Effects:* constructs a new `packaged_task` object and transfers ownership of `other`'s associated task to `*this`, leaving `other` with no associated task.

```
packaged_task& operator=(packaged_task&& other);
```

7 *Effects:* transfers ownerhsip of `other`'s associated task to `*this`, leaving `other` with no associated task. If `*this` had an associated task on entry to this function and that task had not been invoked, sets any futures associated with that task to ready with a `future_error` exception and an error code of `broken_promise` as the result.

8 *Throws:* nothing.

```
~packaged_task();
```

9 *Effects:* destroys `*this`. If `*this` had an associated task and that task had not been invoked, sets any futures associated with that task to ready with a `future_error` exception and an error code of `broken_promise` as the result.

10 *Throws:* nothing.

```
explicit operator bool() const;
```

11 *Returns:* `true` only if `*this` has an associated task.

12 *Throws:* nothing.

```
unique_future<R> get_future();
```

13 *Returns:* a `unique_future` object associated with the result of the associated task of `*this`.

14 *Throws:* `std::bad_function_call` if the future associated with the task has already been retrieved.

```
void operator()(ArgTypes... args);
```

15 *Effects:* *INVOKE*`(f, t1, t2, ..., tN, R)`, where `f` is the associated task of `*this` and `t1, t2, ..., tN` are the values in `args...`. If the task returns normally, the return value is stored as the asynchronous result associated with `*this`, otherwise the exception thrown by the task is stored. Any threads blocked waiting for the asynchronous result associated with the task are unblocked.

16 *Postcondition:* all futures waiting on the asynchronous result are ready.

17 *Throws:* `std::bad_function_call` if the task has already been invoked.

```
void reset();
```

18 *Effects:* returns the object to a state as if a newly-constructed instance had just been assigned to `*this` by `*this = packaged_task(std::move(f))`, where `f` is the associated task of `*this`. If `*this` already had an associated task and that task had not been invoked, sets any futures associated with that task to ready with a `future_error` exception and an error code of `broken_promise` as the result. `get_future()` may now be called again for `*this`.

19 *Postcondition:* `*this` has no associated futures. If `*this` had an associated task, then the new associated task is a copy of the old associated task.

20 *Throws:* `std::bad_alloc` if memory for the internal data structures of the new asynchronous result could not be allcoated.

Annex A (informative) Grammar summary [gram]

1 This summary of C++ syntax is intended to be an aid to comprehension. It is not an exact statement of the language. In particular, the grammar described here accepts a superset of valid C++ constructs. Disambiguation rules (6.8, 7.1, 10.2) must be applied to distinguish expressions from declarations. Further, access control, ambiguity, and type rules must be used to weed out syntactically valid but meaningless constructs.

A.1 Keywords [gram.key]

1 New context-dependent keywords are introduced into a program by `typedef` (7.1.3), namespace (7.3.1), class (clause 9), enumeration (7.2), and `template` (clause 14) declarations.

typedef-name:
    identifier

namespace-name:
    original-namespace-name
    namespace-alias

original-namespace-name:
    identifier

namespace-alias:
    identifier

class-name:
    identifier
    template-id

enum-name:
    identifier

template-name:
    identifier

Note that a *typedef-name* naming a class is also a *class-name* (9.1).

A.2 Lexical conventions [gram.lex]

hex-quad:
    hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

universal-character-name:
    `\u` *hex-quad*
    `\U` *hex-quad hex-quad*

preprocessing-token:
header-name
identifier
pp-number
character-literal
user-defined-character-literal
string-literal
user-defined-string-literal
preprocessing-op-or-punc
each non-white-space character that cannot be one of the above

token:
identifier
keyword
literal
operator
punctuator

header-name:
`<` *h-char-sequence* `>`
`"` *q-char-sequence* `"`

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any member of the source character set except new-line and >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any member of the source character set except new-line and "

pp-number:
digit
`.` *digit*
pp-number digit
pp-number identifier-nondigit
pp-number `e` *sign*
pp-number `E` *sign*
pp-number `.`

identifier:
identifier-nondigit
identifier identifier-nondigit
identifier digit

identifier-nondigit:
nondigit
universal-character-name
other implementation-defined characters

nondigit: one of
`a b c d e f g h i j k l m`
`n o p q r s t u v w x y z`
`A B C D E F G H I J K L M`
`N O P Q R S T U V W X Y Z _`

digit: one of

```
0 1 2 3 4 5 6 7 8 9
```

preprocessing-op-or-punc: one of

```
{     }         [         ]         #         ##        (         )
<:    :>        <%        %>        %:        %:%:      ;         :         ...
new   delete    ?         ::        .         .*
+     -         *         /         %         ^         &         |         ~
!     =         <         >         +=        -=        *=        /=        %=
^=    &=        |=        <<        >>        >>=       <<=       ==        !=
<=    >=        &&        ||        ++        --        ,         ->*       ->
and   and_eq    bitand    bitor     compl     not       not_eq
or    or_eq     xor       xor_eq
```

literal:

integer-literal
character-literal
floating-literal
string-literal
boolean-literal
pointer-literal
user-defined-literal

integer-literal:

decimal-literal integer-suffix$_{opt}$
octal-literal integer-suffix$_{opt}$
hexadecimal-literal integer-suffix$_{opt}$

decimal-literal:

nonzero-digit
decimal-literal digit

octal-literal:

`0`
octal-literal octal-digit

hexadecimal-literal:

`0x` *hexadecimal-digit*
`0X` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of

```
1 2 3 4 5 6 7 8 9
```

octal-digit: one of

```
0 1 2 3 4 5 6 7
```

hexadecimal-digit: one of

```
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
```

integer-suffix:

unsigned-suffix long-suffix$_{opt}$
unsigned-suffix long-long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$
long-long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of

```
u U
```

long-suffix: one of

```
l L
```

long-long-suffix: one of
`ll LL`

character-literal:
`'` *c-char-sequence* `'`
u`'` *c-char-sequence* `'`
U`'` *c-char-sequence* `'`
L`'` *c-char-sequence* `'`

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set except
the single-quote `'`, backslash `\`, or new-line character
escape-sequence
universal-character-name

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
`\'  \"  \?  \\`
`\a  \b  \f  \n  \r  \t  \v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

floating-literal:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+ -`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`f l F L`

string-literal:
" *s-char-sequence*$_{opt}$ "
u8" *s-char-sequence*$_{opt}$ "
u" *s-char-sequence*$_{opt}$ "
U" *s-char-sequence*$_{opt}$ "
L" *s-char-sequence*$_{opt}$ "
R *raw-string*
u8R *raw-string*
uR *raw-string*
UR *raw-string*
LR *raw-string*

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any member of the source character set except
the double-quote ", backslash \, or new-line character
escape-sequence
universal-character-name

raw-string:
" *d-char-sequence*$_{opt}$ [*r-char-sequence*$_{opt}$] *d-char-sequence*$_{opt}$ "

r-char-sequence:
r-char
r-char-sequence r-char

r-char:
any member of the source character set, except
(1), a backslash \followed by a u or U, or
(2), a right square bracket] followed by the initial *d-char-sequence*
(which may be empty) followed by a double quote ".
universal-character-name

d-char-sequence:
d-char
d-char-sequence d-char

d-char:
any member of the basic source character set except:
space, the left square bracket [, the right square bracket],
and the control characters representing horizontal tab,
vertical tab, form feed, and newline.

boolean-literal:
false
true

pointer-literal:
nullptr

user-defined-literal:
user-defined-integer-literal
user-defined-floating-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix

user-defined-floating-literal:
fractional-constant exponent-part$_{opt}$ *ud-suffix*
digit-sequence exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix

user-defined-character-literal:
character-literal ud-suffix

ud-suffix:
identifier

A.3 Basic concepts [gram.basic]

translation-unit:
declaration-seq$_{opt}$

A.4 Expressions [gram.expr]

primary-expression:
literal
`this`
`(` *expression* `)`
id-expression
lambda-expression
id-expression:
unqualified-id
qualified-id
unqualified-id:
identifier
operator-function-id
conversion-function-id
literal-operator-id
$\sim$ *class-name*
template-id

qualified-id:
`::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *unqualified-id*
`::` *identifier*
`::` *operator-function-id*
`::` *template-id*

nested-name-specifier:
type-name `::`
namespace-name `::`
nested-name-specifier identifier `::`
nested-name-specifier `template`$_{opt}$ *simple-template-id* `::`
nested-name-specifier$_{opt}$ *concept-id* `::`

lambda-expression:
lambda-introducer lambda-parameter-declaration$_{opt}$ *compound-statement*

lambda-introducer:
`[` *lambda-capture*$_{opt}$ `]`

lambda-capture:
capture-default
capture-list
capture-default , *capture-list*

capture-default:
&
=

capture-list:
capture
capture-list , *capture*

capture:
identifier
& *identifier*
this

lambda-parameter-declaration:
(*lambda-parameter-declaration-list*$_{opt}$) mutable$_{opt}$ *attribute-specifier*$_{opt}$
exception-specification$_{opt}$ *lambda-return-type-clause*$_{opt}$

lambda-parameter-declaration-list:
lambda-parameter
lambda-parameter , *lambda-parameter-declaration-list*

lambda-parameter:
decl-specifier-seq attribute-specifier$_{opt}$ *declarator*

lambda-return-type-clause:
-> *attribute-specifier*$_{opt}$ *type-id*

postfix-expression:
primary-expression
postfix-expression [*expression*]
postfix-expression [*braced-init-list*]
postfix-expression (*expression-list*$_{opt}$)
simple-type-specifier (*expression-list*$_{opt}$)
typename-specifier (*expression-list*$_{opt}$)
simple-type-specifier braced-init-list
typename-specifier braced-init-list
postfix-expression . template$_{opt}$ *id-expression*
postfix-expression -> template$_{opt}$ *id-expression*
postfix-expression . *pseudo-destructor-name*
postfix-expression -> *pseudo-destructor-name*
postfix-expression ++
postfix-expression --
dynamic_cast < *type-id* > (*expression*)
static_cast < *type-id* > (*expression*)
reinterpret_cast < *type-id* > (*expression*)
const_cast < *type-id* > (*expression*)
typeid (*expression*)
typeid (*type-id*)

expression-list:
initializer-list

pseudo-destructor-name:
::$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name* :: ~ *type-name*
::$_{opt}$ *nested-name-specifier* template *simple-template-id* :: ~ *type-name*
::$_{opt}$ *nested-name-specifier*$_{opt}$ ~ *type-name*

unary-expression:
postfix-expression
`++` *cast-expression*
`--` *cast-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
new-expression
delete-expression

unary-operator: one of
`*` *&* `+` `-` *!* ∼

new-expression:
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initializer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initializer*$_{opt}$

new-placement:
`(` *expression-list* `)`

new-type-id:
type-specifier-seq new-declarator$_{opt}$

new-declarator:
ptr-operator new-declarator$_{opt}$
noptr-new-declarator

noptr-new-declarator:
`[` *expression* `]`
noptr-new-declarator `[` *constant-expression* `]`

new-initializer:
`(` *expression-list*$_{opt}$ `)`
braced-init-list

delete-expression:
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*

cast-expression:
unary-expression
`(` *type-id* `)` *cast-expression*

pm-expression:
cast-expression
pm-expression `.*` *cast-expression*
pm-expression `->*` *cast-expression*

multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*

relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*

equality-expression:
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

and-expression:
equality-expression
and-expression `&` *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression `^` *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression `|` *exclusive-or-expression*

logical-and-expression:
inclusive-or-expression
logical-and-expression `&&` *inclusive-or-expression*

logical-or-expression:
logical-and-expression
logical-or-expression || *logical-and-expression*

conditional-expression:
logical-or-expression
logical-or-expression `?` *expression* `:` *assignment-expression*

assignment-expression:
conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression

assignment-operator: one of
`= *= /= %= += -= >>= <<= &= ^= |=`

expression:
assignment-expression
expression `,` *assignment-expression*

constant-expression:
conditional-expression

A.5 Statements [gram.stmt]

statement:
labeled-statement
attribute-specifier$_{opt}$ *expression-statement*
attribute-specifier$_{opt}$ *compound-statement*
attribute-specifier$_{opt}$ *selection-statement*
attribute-specifier$_{opt}$ *iteration-statement*
attribute-specifier$_{opt}$ *jump-statement*
declaration-statement
attribute-specifier$_{opt}$ *try-block*
late-checked-block

labeled-statement:
attribute-specifier$_{opt}$ *identifier* `:` *statement*
attribute-specifier$_{opt}$ `case` *constant-expression* `:` *statement*
attribute-specifier$_{opt}$ `default` `:` *statement*

expression-statement:
expression$_{opt}$ `;`

compound-statement:
`{` *statement-seq*$_{opt}$ `}`

statement-seq:
statement
statement-seq statement

selection-statement:
`if` `(` *condition* `)` *statement*
`if` `(` *condition* `)` *statement* `else` *statement*
`switch` `(` *condition* `)` *statement*

condition:
expression
type-specifier-seq attribute-specifier$_{opt}$ *declarator* `=` *initializer-clause*
type-specifier-seq attribute-specifier$_{opt}$ *declarator braced-init-list*

iteration-statement:
`while` `(` *condition* `)` *statement*
`do` *statement* `while` `(` *expression* `)` `;`
`for` `(` *for-init-statement condition*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
`for` `(` *for-range-declaration* `:` *expression* `)` *statement*

for-init-statement:
expression-statement
simple-declaration

for-range-declaration:
type-specifier-seq attribute-specifier$_{opt}$ *declarator*

jump-statement:
`break` `;`
`continue` `;`
`return` *expression*$_{opt}$ `;`
`return` *braced-init-list* `;`
`goto` *identifier* `;`

declaration-statement:
block-declaration

late-checked-block:
`late_check` *compound-statement*

A.6 Declarations [gram.dcl]

declaration-seq:
declaration
declaration-seq declaration

declaration:
block-declaration
function-definition
template-declaration
explicit-instantiation
explicit-specialization
linkage-specification
namespace-definition
concept-definition
concept-map-definition
attribute-declaration

block-declaration:
simple-declaration
asm-definition
namespace-alias-definition
using-declaration
using-directive
static_assert-declaration
alias-declaration
opaque-enum-declaration

alias-declaration:
`using` *identifier* = *type-id* `;`

simple-declaration:
attribute-specifier$_{opt}$ *decl-specifier-seq*$_{opt}$ *attribute-specifier*$_{opt}$ *init-declarator-list*$_{opt}$ `;`

static_assert-declaration:
`static_assert (` *constant-expression* `,` *string-literal* `) ;`

attribute-declaration:
attribute-specifier `;`

decl-specifier:
storage-class-specifier
type-specifier
function-specifier
`friend`
`typedef`
`constexpr`
alignment-specifier

decl-specifier-seq:
decl-specifier-seq$_{opt}$ *decl-specifier*

storage-class-specifier:
`register`
`static`
`thread_local`
`extern`
`mutable`

function-specifier:
`inline`
`virtual`
`explicit`

typedef-name:
identifier

type-specifier:
simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
typename-specifier
cv-qualifier

type-specifier-seq:
type-specifier type-specifier-seq$_{opt}$

simple-type-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *type-name*
`::`$_{opt}$ *nested-name-specifier* `template` *simple-template-id*
`char`
`char16_t`
`char32_t`
`wchar_t`
`bool`
`short`
`int`
`long`
`signed`
`unsigned`
`float`
`double`
`void`
`auto`
`decltype (` *expression* `)`

type-name:
class-name
enum-name
typedef-name

elaborated-type-specifier:
class-key `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*
class-key `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `template`$_{opt}$ *simple-template-id*
enum `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *identifier*

enum-name:
identifier

enum-specifier:
enum-head `{` *enumerator-list*$_{opt}$ `}`
enum-head `{` *enumerator-list* `,` `}`

enum-head:
enum-key identifier$_{opt}$ *attribute-specifier*$_{opt}$ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$
enum-key nested-name-specifier identifier
attribute-specifier$_{opt}$ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$

opaque-enum-declaration:
enum-key identifier attribute-specifier$_{opt}$ *enum-base*$_{opt}$ *attribute-specifier*$_{opt}$ `;`

enum-key:
`enum`
`enum class`
`enum struct`

enum-base:
`:` *type-specifier-seq*

enumerator-list:
enumerator-definition
enumerator-list `,` *enumerator-definition*

enumerator-definition:
enumerator
enumerator `=` *constant-expression*

enumerator:
identifier

namespace-name:
original-namespace-name
namespace-alias

original-namespace-name:
identifier

namespace-definition:
named-namespace-definition
unnamed-namespace-definition

named-namespace-definition:
original-namespace-definition
extension-namespace-definition

original-namespace-definition:
`inline`$_{opt}$ `namespace` *identifier* `{` *namespace-body* `}`

extension-namespace-definition:
`inline`$_{opt}$ `namespace` *original-namespace-name* `{` *namespace-body* `}`

unnamed-namespace-definition:
`inline`$_{opt}$ `namespace` `{` *namespace-body* `}`

namespace-body:
declaration-seq$_{opt}$

namespace-alias:
identifier

namespace-alias-definition:
`namespace` *identifier* `=` *qualified-namespace-specifier* `;`

qualified-namespace-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name*

using-declaration:
`using` `typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using` `::` *unqualified-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name*$_{opt}$ `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `;`

using-directive:
attribute-specifier$_{opt}$ `using` `namespace` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *namespace-name* `;`

asm-definition:
`asm` `(` *string-literal* `)` `;`

linkage-specification:
`extern` *string-literal* `{` *declaration-seq*$_{opt}$ `}`
`extern` *string-literal declaration*

attribute-specifier:
`[ [` *attribute-list* `] ]`

attribute-list:
attribute$_{opt}$
attribute-list `,` *attribute*$_{opt}$

attribute:
attribute-token attribute-argument-clause$_{opt}$

attribute-token:
identifier
attribute-scoped-token

attribute-scoped-token:
attribute-namespace `::` *identifier*

attribute-namespace:
identifier

attribute-argument-clause:
`(` *balanced-token-seq* `)`

balanced-token-seq:
balanced-token
balanced-token-seq balanced-token

balanced-token:
`(` *balanced-token-seq* `)`
`[` *balanced-token-seq* `]`
`{` *balanced-token-seq* `}`
any *token* other than a parenthesis, a bracket, or a brace

A.7 Declarators [gram.decl]

init-declarator-list:
init-declarator
init-declarator-list `,` *init-declarator*

init-declarator:
declarator initializer$_{opt}$

declarator:
ptr-declarator
noptr-declarator parameters-and-qualifiers `->` *attribute-specifier*$_{opt}$ *type-id*

ptr-declarator:
noptr-declarator
ptr-operator ptr-declarator

noptr-declarator:
declarator-id attribute-specifier$_{opt}$
noptr-declarator parameters-and-qualifiers
noptr-declarator `[` *constant-expression*$_{opt}$ `]` *attribute-specifier*$_{opt}$
`(` *ptr-declarator* `)`

parameters-and-qualifiers:
`(` *parameter-declaration-clause* `)` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
ref-qualifier$_{opt}$ *exception-specification*$_{opt}$

ptr-operator:
`*` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$
`&`
`&&`
`::`$_{opt}$ *nested-name-specifier* `*` *attribute-specifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$

cv-qualifier-seq:
cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:
`const`
`volatile`

ref-qualifier:
`&`
`&&`

declarator-id:
`...`$_{opt}$ *id-expression*
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*

type-id:
type-specifier-seq attribute-specifier$_{opt}$ *abstract-declarator*$_{opt}$

abstract-declarator:
ptr-abstract-declarator
noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers* `->` *attribute-specifier*$_{opt}$ *type-id*
`...`

ptr-abstract-declarator:
noptr-abstract-declarator
ptr-operator ptr-abstract-declarator$_{opt}$

noptr-abstract-declarator:
noptr-abstract-declarator$_{opt}$ *parameters-and-qualifiers*
noptr-abstract-declarator$_{opt}$ `[` *constant-expression* `]` *attribute-specifier*$_{opt}$
`(` *ptr-abstract-declarator* `)`

parameter-declaration-clause:
parameter-declaration-list$_{opt}$ *...*$_{opt}$
parameter-declaration-list `,` *...*

parameter-declaration-list:
parameter-declaration
parameter-declaration-list `,` *parameter-declaration*

parameter-declaration:
decl-specifier-seq attribute-specifier$_{opt}$ *declarator*
decl-specifier-seq attribute-specifier$_{opt}$ *declarator* `=` *assignment-expression*
decl-specifier-seq attribute-specifier$_{opt}$ *abstract-declarator*$_{opt}$
decl-specifier-seq attribute-specifier$_{opt}$ *abstract-declarator*$_{opt}$ `=` *assignment-expression*

function-definition:
decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator function-body*
decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= default ;`
decl-specifier-seq$_{opt}$ *attribute-specifier*$_{opt}$ *declarator* `= delete ;`

function-body:
ctor-initializer$_{opt}$ *compound-statement*
function-try-block

initializer:
brace-or-equal-initializer
`(` *expression-list* `)`

brace-or-equal-initializer:
`=` *initializer-clause*
braced-init-list

initializer-clause:
assignment-expression
braced-init-list

initializer-list:
initializer-clause ...$_{opt}$
initializer-list , *initializer-clause* ...$_{opt}$

braced-init-list:
{ *initializer-list* ,$_{opt}$ }
{ }

A.8 Classes [gram.class]

class-name:
identifier
simple-template-id

class-specifier:
class-head { *member-specification*$_{opt}$ }

class-head:
class-key identifier$_{opt}$ *attribute-specifier*$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier identifier attribute-specifier$_{opt}$ *base-clause*$_{opt}$
class-key nested-name-specifier$_{opt}$ *simple-template-id attribute-specifier*$_{opt}$ *base-clause*$_{opt}$

class-key:
class
struct
union

member-specification:
member-declaration member-specification$_{opt}$
access-specifier : *member-specification*$_{opt}$

member-declaration:
member-requirement$_{opt}$ *decl-specifier-seq*$_{opt}$
attribute-specifier$_{opt}$ *member-declarator-list*$_{opt}$;
member-requirement$_{opt}$ *function-definition* ;$_{opt}$
::$_{opt}$ *nested-name-specifier* template$_{opt}$ *unqualified-id* ;
using-declaration
static_assert-declaration
template-declaration

member-requirement:
requires-clause

member-declarator-list:
member-declarator
member-declarator-list , *member-declarator*

member-declarator:
declarator pure-specifier$_{opt}$
declarator brace-or-equal-initializer$_{opt}$
identifier$_{opt}$ *attribute-specifier*$_{opt}$: *constant-expression*

pure-specifier:
= 0

A.9 Derived classes [gram.derived]

base-clause:
: *base-specifier-list*

base-specifier-list:
base-specifier `...`$_{opt}$
base-specifier-list `,` *base-specifier* `...`$_{opt}$

base-specifier:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name attribute-specifier*$_{opt}$
`virtual` *access-specifier*$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name attribute-specifier*$_{opt}$
access-specifier `virtual`$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name attribute-specifier*$_{opt}$

access-specifier:
`private`
`protected`
`public`

A.10 Special member functions [gram.special]

conversion-function-id:
`operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq attribute-specifier$_{opt}$ *conversion-declarator*$_{opt}$

conversion-declarator:
ptr-operator conversion-declarator$_{opt}$

ctor-initializer:
`:` *mem-initializer-list*

mem-initializer-list:
mem-initializer `...`$_{opt}$
mem-initializer `,` *mem-initializer-list* `...`$_{opt}$

mem-initializer:
mem-initializer-id `(` *expression-list*$_{opt}$ `)`
mem-initializer-id braced-init-list

mem-initializer-id:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *class-name*
identifier

A.11 Overloading [gram.over]

operator-function-id:
`operator` *operator*

operator: one of

`new`	`delete`	`new[]`	`delete[]`					
`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`	`~`
`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`	`<<=`	`==`	`!=`
`<=`	`>=`	`&&`	`||`	`++`	`--`	`,`	`->*`	`->`
`()`	`[]`							

literal-operator-id:
`operator ""` *identifier*

A.12 Templates [gram.temp]

template-declaration:
`export`$_{opt}$ `template <` *template-parameter-list* `>` *declaration*

template-parameter-list:
template-parameter
template-parameter-list , *template-parameter*

template-parameter:
type-parameter
parameter-declaration
constrained-template-parameter

type-parameter:
`class` ...$_{opt}$ *identifier*$_{opt}$
`class` *identifier*$_{opt}$ `=` *type-id*
`typename` ...$_{opt}$ *identifier*$_{opt}$
`typename` *identifier*$_{opt}$ `=` *type-id*
`template` `<` *template-parameter-list* `>` `class` ...$_{opt}$ *identifier*$_{opt}$
`template` `<` *template-parameter-list* `>` `class` *identifier*$_{opt}$ `=` *id-expression*

constrained-template-parameter:
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* ...$_{opt}$ *identifier*$_{opt}$
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name identifier*$_{opt}$ *constrained-default-argument*$_{opt}$
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `<`
simple-requirement-argument-list `>` ...$_{opt}$ *identifier*
`::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name* `<`
simple-requirement-argument-list `>` *identifier constrained-default-argument*$_{opt}$

constrained-default-argument:
`=` *type-id*
`=` *assignment-expression*
`=` *id-expression*

simple-requirement-argument-list:
`auto`
`auto` , *template-argument-list*

simple-template-id:
template-name `<` *template-argument-list*$_{opt}$ `>`

template-id:
simple-template-id
operator-function-id `<` *template-argument-list*$_{opt}$ `>`

template-name:
identifier

template-argument-list:
template-argument ...$_{opt}$
template-argument-list , *template-argument* ...$_{opt}$

template-argument:
constant-expression
type-id
id-expression

typename-specifier:
`typename` `::`$_{opt}$ *nested-name-specifier identifier*
`typename` `::`$_{opt}$ *nested-name-specifier* `template`$_{opt}$ *simple-template-id*

explicit-instantiation:
`extern`$_{opt}$ `template` *declaration*

explicit-specialization:
`template` `<` `>` *declaration*

concept-id:
concept-name `<` *template-argument-list*$_{opt}$ `>`

concept-name:
identifier

concept-definition:
`auto`$_{opt}$ `concept` *identifier* `<` *template-parameter-list* `>`
refinement-clause$_{opt}$ *concept-body* `;`$_{opt}$

concept-body:
`{` *concept-member-specification*$_{opt}$ `}`

concept-member-specification:
concept-member-specifier concept-member-specification$_{opt}$

concept-member-specifier:
associated-function
type-parameter `;`
associated-requirements
axiom-definition

associated-function:
simple-declaration
function-definition
template-declaration

associated-requirements:
requires-clause `;`

axiom-definition:
requires-clause$_{opt}$ `axiom` *identifier* `(` *parameter-declaration-clause* `)` *axiom-body*

axiom-body:
`{` *axiom-seq*$_{opt}$ `}`

axiom-seq:
axiom axiom-seq$_{opt}$

axiom:
expression-statement
`if` `(` *expression* `)` *expression-statement*

concept-map-definition:
`concept_map` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id* `{` *concept-map-member-specification*$_{opt}$ `}` `;`$_{opt}$

concept-map-member-specification:
concept-map-member concept-map-member-specification$_{opt}$

concept-map-member:
simple-declaration
function-definition
template-declaration

refinement-clause:
`:` *refinement-specifier-list*

refinement-specifier-list:
refinement-specifier `,` *refinement-specifier-list*
refinement-specifier

refinement-specifier:
concept-instance-alias-def$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

concept-instance-alias-def:
identifier `=`

requires-clause:
`requires` *requirement-list*
`requires (` *requirement-list* `)`

requirement-list:
requirement `...`$_{opt}$ `&&` *requirement-list*
requirement `...`$_{opt}$

requirement:
concept-instance-alias-def$_{opt}$ `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
`!` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

A.13 Exception handling [gram.except]

try-block:
`try` *compound-statement handler-seq*

function-try-block:
`try` *ctor-initializer*$_{opt}$ *compound-statement handler-seq*

handler-seq:
handler handler-seq$_{opt}$

handler:
`catch (` *exception-declaration* `)` *compound-statement*

exception-declaration:
type-specifier-seq declarator
type-specifier-seq abstract-declarator
type-specifier-seq
`...`

throw-expression:
`throw` *assignment-expression*$_{opt}$

exception-specification:
`throw (` *type-id-list*$_{opt}$ `)`

type-id-list:
type-id `...`$_{opt}$
type-id-list `,` *type-id* `...`$_{opt}$

A.14 Preprocessing directives [gram.cpp]

preprocessing-file:
group$_{opt}$

group:
group-part
group group-part

group-part:
if-section
control-line
text-line
 # non-directive

if-section:
if-group elif-groups$_{opt}$ *else-group*$_{opt}$ *endif-line*

if-group:
`# if` *constant-expression new-line* $group_{opt}$
`# ifdef` *identifier new-line* $group_{opt}$
`# ifndef` *identifier new-line* $group_{opt}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
`# elif` *constant-expression new-line* $group_{opt}$

else-group:
`# else` *new-line* $group_{opt}$

endif-line:
`# endif` *new-line*

control-line:
`# include` *pp-tokens new-line*
`# define` *identifier replacement-list new-line*
`# define` *identifier lparen identifier-list*$_{opt}$ `)` *replacement-list new-line*
`# define` *identifier lparen* `...` `)` *replacement-list new-line*
`# define` *identifier lparen identifier-list,* `...` `)` *replacement-list new-line*
`# undef` *identifier new-line*
`# line` *pp-tokens new-line*
`# error` $pp\text{-}tokens_{opt}$ *new-line*
`# pragma` $pp\text{-}tokens_{opt}$ *new-line*
`#` *new-line*

text-line:
$pp\text{-}tokens_{opt}$ *new-line*

non-directive:
$pp\text{-}tokens_{opt}$ *new-line*

lparen:
a `(` character not immediately preceded by white-space

identifier-list:
identifier
identifier-list `,` *identifier*

replacement-list:
$pp\text{-}tokens_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

new-line:
the new-line character

Annex B (informative) Implementation quantities [implimits]

1 Because computers are finite, C++ implementations are inevitably limited in the size of the programs they can successfully process. Every implementation shall document those limitations where known. This documentation may cite fixed limits where they exist, say how to compute variable limits as a function of available resources, or say that fixed limits do not exist or are unknown.

2 The limits may constrain quantities that include those described below or others. The bracketed number following each quantity is recommended as the minimum for that quantity. However, these quantities are only guidelines and do not determine compliance.

- Nesting levels of compound statements, iteration control structures, and selection control structures [256].
- Nesting levels of conditional inclusion [256].
- Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration [256].
- Nesting levels of parenthesized expressions within a full-expression [256].
- Number of characters in an internal identifier or macro name [1 024].
- Number of characters in an external identifier [1 024].
- External identifiers in one translation unit [65 536].
- Identifiers with block scope declared in one block [1 024].
- Macro identifiers simultaneously defined in one translation unit [65 536].
- Parameters in one function definition [256].
- Arguments in one function call [256].
- Parameters in one macro definition [256].
- Arguments in one macro invocation [256].
- Characters in one logical source line [65 536].
- Characters in a character string literal or wide string literal (after concatenation) [65 536].
- Size of an object [262 144].
- Nesting levels for `#include` files [256].
- Case labels for a `switch` statement (excluding those for any nested `switch` statements) [16 384].
- Data members in a single class [16 384].
- Enumeration constants in a single enumeration [4 096].
- Levels of nested class definitions in a single *member-specification* [256].
- Functions registered by `atexit()` [32].

— Direct and indirect base classes [16 384].
— Direct base classes for a single class [1 024].
— Members declared in a single class [4 096].
— Final overriding virtual functions in a class, accessible or not [16 384].
— Direct and indirect virtual bases of a class [1 024].
— Static members of a class [1 024].
— Friend declarations in a class [4 096].
— Access control declarations in a class [4 096].
— Member initializers in a constructor definition [6 144].
— Scope qualifications of one identifier [256].
— Nested external specifications [1 024].
— Template arguments in a template declaration [1 024].
— Recursively nested template instantiations [17].
— Recursively nested implicit concept map definitions [1 024].
— Handlers per `try` block [256].
— Throw specifications on a single function declaration [256].

Annex C (informative) Compatibility [diff]

C.1 C++ and ISO C [diff.iso]

1 The subclauses of this subclause list the differences between C++ and ISO C, by the chapters of this document.

C.1.1 Clause 2: lexical conventions [diff.lex]

2.3
Change: C++ style comments (`//`) are added
A pair of slashes now introduce a one-line comment.
Rationale: This style of comments is a useful addition to the language.
Effect on original feature: Change to semantics of well-defined feature. A valid ISO C expression containing a division operator followed immediately by a C-style comment will now be treated as a C++ style comment. For example:

```
int a = 4;
int b = 8                    //* divide by a*/ a;
+a;
```

Difficulty of converting: Syntactic transformation. Just add white space after the division operator.
How widely used: The token sequence `//*` probably occurs very seldom.

2.11
Change: New Keywords
New keywords are added to C++; see 2.11.
Rationale: These keywords were added in order to implement the new semantics of C++.
Effect on original feature: Change to semantics of well-defined feature. Any ISO C programs that used any of these keywords as identifiers are not valid C++ programs.
Difficulty of converting: Syntactic transformation. Converting one specific program is easy. Converting a large collection of related programs takes more work.
How widely used: Common.

2.13.2
Change: Type of character literal is changed from `int` to `char`
Rationale: This is needed for improved overloaded function argument type matching. For example:

```
int function( int i );
int function( char c );

function( 'x' );
```

It is preferable that this call match the second version of function rather than the first.
Effect on original feature: Change to semantics of well-defined feature. ISO C programs which depend on

```
sizeof('x') == sizeof(int)
```

will not work the same as C++ programs.
Difficulty of converting: Simple.
How widely used: Programs which depend upon `sizeof('x')` are probably rare.

Subclause 2.13.4:
Change: String literals made const
The type of a string literal is changed from "array of `char` " to "array of `const char`." The type of a `char16_t` string literal is changed from "array of *some-integer-type* " to "array of `const char16_t`." The type of a `char32_t` string literal is changed from "array of *some-integer-type* " to "array of `const char32_t`." The type of a wide string literal is changed from "array of `wchar_t` " to "array of `const wchar_t`."
Rationale: This avoids calling an inappropriate overloaded function, which might expect to be able to modify its argument.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Simple syntactic transformation, because string literals can be converted to `char*`; (4.2). The most common cases are handled by a new but deprecated standard conversion:

```
char* p = "abc";                // valid in C, deprecated in C++
char* q = expr ? "abc" : "de";  // valid in C, invalid in C++
```

How widely used: Programs that have a legitimate reason to treat string literals as pointers to potentially modifiable memory are probably rare.

C.1.2 Clause 3: basic concepts [diff.basic]

3.1
Change: C++ does not have "tentative definitions" as in C E.g., at file scope,

```
int i;
int i;
```

is valid in C, invalid in C++. This makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X *next; };

static struct X a;
static struct X b = { 0, &a };
static struct X a = { 1, &b };
```

Rationale: This avoids having different initialization rules for built-in types and user-defined types.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
Rationale: In C++, the initializer for one of a set of mutually-referential file-local static objects must invoke a function call to achieve the initialization.
How widely used: Seldom.

3.3
Change: A `struct` is a scope in C++, not in C
Rationale: Class scope is crucial to C++, and a struct is a class.
Effect on original feature: Change to semantics of well-defined feature.

Difficulty of converting: Semantic transformation.
How widely used: C programs use `struct` extremely frequently, but the change is only noticeable when `struct`, enumeration, or enumerator names are referred to outside the `struct`. The latter is probably rare.

3.5 [also 7.1.6]
Change: A name of file scope that is explicitly declared `const`, and not explicitly declared `extern`, has internal linkage, while in C it would have external linkage
Rationale: Because `const` objects can be used as compile-time values in C++, this feature urges programmers to provide explicit initializer values for each `const`. This feature allows the user to put `const`objects in header files that are included in many compilation units.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation
How widely used: Seldom

3.6
Change: Main cannot be called recursively and cannot have its address taken
Rationale: The main function may require special actions.
Effect on original feature: Deletion of semantically well-defined feature
Difficulty of converting: Trivial: create an intermediary function such as `mymain(argc, argv`.
How widely used: Seldom

3.9
Change: C allows "compatible types" in several places, C++ does not For example, otherwise-identical `struct` types with different tag names are "compatible" in C but are distinctly different types in C++.
Rationale: Stricter type checking is essential for C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The "typesafe linkage" mechanism will find many, but not all, of such problems. Those problems not found by typesafe linkage will continue to function properly, according to the "layout compatibility rules" of this International Standard.
How widely used: Common.

4.10
Change: Converting `void*` to a pointer-to-object type requires casting

```
char a[10];
void *b=a;
void foo() {
  char *c=b;
}
```

ISO C will accept this usage of pointer to void being assigned to a pointer to object type. C++ will not.
Rationale: C++ tries harder than C to enforce compile-time type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. Violations will be diagnosed by the C++ translator. The fix is to add a cast For example:

```
char *c = (char *) b;
```

How widely used: This is fairly widely used but it is good programming practice to add the cast when assigning pointer-to-void to pointer-to-object. Some ISO C translators will give a warning if the cast is not used.

4.10
Change: Only pointers to non-const and non-volatile objects may be implicitly converted to `void*`

Rationale: This improves type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Could be automated. A C program containing such an implicit conversion from (e.g.) pointer-to-const-object to void* will receive a diagnostic message. The correction is to add an explicit cast.
How widely used: Seldom.

C.1.3 Clause 5: expressions [diff.expr]

5.2.2
Change: Implicit declaration of functions is not allowed
Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature. Note: the original feature was labeled as "obsolescent" in ISO C.
Difficulty of converting: Syntactic transformation. Facilities for producing explicit function declarations are fairly widespread commercially.
How widely used: Common.

5.3.3, 5.4
Change: Types must be declared in declarations, not in expressions In C, a sizeof expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x .
Rationale: This prohibition helps to clarify the location of declarations in the source code.
Effect on original feature: Deletion of a semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

5.16, 5.17, 5.18

Change: The result of a conditional expression, an assignment expression, or a comma expression may be an lvalue
Rationale: C++ is an object-oriented language, placing relatively more emphasis on lvalues. For example, functions may return lvalues.
Effect on original feature: Change to semantics of well-defined feature. Some C expressions that implicitly rely on lvalue-to-rvalue conversions will yield different results. For example,

```
char arr[100];
sizeof(0, arr)
```

yields `100` in C++ and `sizeof(char*)` in C.
Difficulty of converting: Programs must add explicit casts to the appropriate rvalue.
How widely used: Rare.

C.1.4 Clause 6: statements [diff.stat]

6.4.2, 6.6.4 (`switch` and `goto` statements)
Change: It is now invalid to jump past a declaration with explicit or implicit initializer (except across entire block not entered)
Rationale: Constructors used in initializers may allocate resources which need to be de-allocated upon leaving the block. Allowing jump past initializers would require complicated run-time determination of

allocation. Furthermore, any use of the uninitialized object could be a disaster. With this simple compile-time rule, C++ assures that if an initialized variable is in scope, then it has assuredly been initialized.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

6.6.3
Change: It is now invalid to return (explicitly or implicitly) from a function which is declared to return a value without actually returning a value
Rationale: The caller and callee may assume fairly elaborate return-value mechanisms for the return of class objects. If some flow paths execute a return without specifying any value, the implementation must embody many more complications. Besides, promising to return a value of a given type, and then not returning such a value, has always been recognized to be a questionable practice, tolerated only because very-old C had no distinction between void functions and int functions.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Add an appropriate return value to the source code, e.g. zero.
How widely used: Seldom. For several years, many existing C implementations have produced warnings in this case.

C.1.5 Clause 7: declarations [diff.dcl]

7.1.1
Change: In C++, the `static` or `extern` specifiers can only be applied to names of objects or functions Using these specifiers with type declarations is illegal in C++. In C, these specifiers are ignored when used on type declarations.

Example:

```
static struct S {               // valid C, invalid in C++
  int i;
};
```

Rationale: Storage class specifiers don't have any meaning when associated with a type. In C++, class members can be declared with the `static` storage class specifier. Allowing storage class specifiers on type declarations could render the code confusing for users.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

7.1.3
Change: A C++ typedef name must be different from any class type name declared in the same scope (except if the typedef is a synonym of the class name with the same name). In C, a typedef name and a struct tag name declared in the same scope can have the same name (because they have different name spaces)

Example:

```
typedef struct name1 { /*...*/ } name1;          // valid C and C++
struct name { /*...*/ };
typedef int name;                // valid C, invalid C++
```

Rationale: For ease of use, C++ doesn't require that a type name be prefixed with the keywords `class`, `struct` or `union` when used in object declarations or type casts.

Example:

```
class name { /*...*/ };
name i;                          // i has type class name
```

Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. One of the 2 types has to be renamed.
How widely used: Seldom.

7.1.6 [see also 3.5]
Change: const objects must be initialized in C++ but can be left uninitialized in C
Rationale: A const object cannot be assigned to so it must be initialized to hold a useful value.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom.

7.1.6 (type specifiers)
Change: Banning implicit int

In C++ a *decl-specifier-seq* must contain a *type-specifier*. In the following example, the left-hand column presents valid C; the right-hand column presents equivalent C++:

```
void f(const parm);              void f(const int parm);
const n = 3;                     const int n = 3;
main()                           int main()
    /* ... */                       /* ... */
```

Rationale: In C++, implicit int creates several opportunities for ambiguity between expressions involving function-like casts and declarations. Explicit declaration is increasingly considered to be proper style. Liaison with WG14 (C) indicated support for (at least) deprecating implicit int in the next revision of C.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. Could be automated.
How widely used: Common.

7.1.6.4
Change: The keyword `auto` cannot be used as a storage class specifier.

```
void f() {
  auto int x;                    // valid C, invalid C++
}
```

Rationale: Allowing the use of `auto` to deduce the type of a variable from its initializer results in undesired interpretations of `auto` as a storage class specifier in certain contexts.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Rare.

7.2
Change: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type

Example:

```
enum color { red, blue, green };
enum color c = 1;               // valid C, invalid C++
```

Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)
How widely used: Common.

7.2
Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is `int`.

Example:

```
enum e { A };
sizeof(A) == sizeof(int)        // in C
sizeof(A) == sizeof(e)          // in C++
/* and sizeof(int) is not necessarily equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

C.1.6 Clause 8: declarators [diff.decl]

8.3.5
Change: In C++, a function declared with an empty parameter list takes no arguments. In C, an empty parameter list means that the number and type of the function arguments are unknown"

Example:

```
int f();            // means int f(void) in C++
                                        // int f( unknown ) in C
```

Rationale: This is to avoid erroneous function calls (i.e. function calls with the wrong number or type of arguments).
Effect on original feature: Change to semantics of well-defined feature. This feature was marked as "obsolescent" in C.
Difficulty of converting: Syntactic transformation. The function declarations using C incomplete declaration style must be completed to become full prototype declarations. A program may need to be updated further if different calls to the same (non-prototype) function have different numbers of arguments or if the type of corresponding arguments differed.
How widely used: Common.

8.3.5 [see 5.3.3]
Change: In C++, types may not be defined in return or parameter types. In C, these type definitions are allowed

Example:

```
void f( struct S { int a; } arg ) {}     // valid C, invalid C++
enum E { A, B, C } f() {}                // valid C, invalid C++
```

Rationale: When comparing types in different compilation units, C++ relies on name equivalence when C relies on structural equivalence. Regarding parameter types: since the type defined in an parameter list would be in the scope of the function, the only legal calls in C++ would be from within the function itself.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The type definitions must be moved to file scope, or in header files.
How widely used: Seldom. This style of type definitions is seen as poor coding style.

8.4
Change: In C++, the syntax for function definition excludes the "old-style" C function. In C, "old-style" syntax is allowed, but deprecated as "obsolescent."
Rationale: Prototypes are essential to type safety.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Common in old programs, but already known to be obsolescent.

8.5.2
Change: In C++, when initializing an array of character with a string, the number of characters in the string (including the terminating `'\0'`) must not exceed the number of elements in the array. In C, an array can be initialized with a string even if the array is not large enough to contain the string-terminating `'\0'`

Example:

```
char array[4] = "abcd";          // valid C, invalid C++
```

Rationale: When these non-terminated arrays are manipulated by standard string routines, there is potential for major catastrophe.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. The arrays must be declared one element bigger to contain the string terminating `'\0'`.
How widely used: Seldom. This style of array initialization is seen as poor coding style.

C.1.7 Clause 9: classes [diff.class]

9.1 [see also 7.1.3]
Change: In C++, a class declaration introduces the class name into the scope where it is declared and hides any object, function or other declaration of that name in an enclosing scope. In C, an inner scope declaration of a struct tag name never hides the name of an object or function in an outer scope

Example:

```
int x[99];
void f() {
        struct x { int a; };
        sizeof(x);  /* size of the array in C */
        /* size of the struct in C++ */
}
```

Rationale: This is one of the few incompatibilities between C and C++ that can be attributed to the new

C++ name space definition where a name can be declared as a type and as a non-type in a single scope causing the non-type name to hide the type name and requiring that the keywords `class`, `struct`, `union` or `enum` be used to refer to the type name. This new name space definition provides important notational conveniences to C++ programmers and helps making the use of the user-defined types as similar as possible to the use of built-in types. The advantages of the new name space definition were judged to outweigh by far the incompatibility with C described above.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation. If the hidden name that needs to be accessed is at global scope, the `::` C++ operator can be used. If the hidden name is at block scope, either the type or the struct tag has to be renamed.
How widely used: Seldom.

9.7
Change: In C++, the name of a nested class is local to its enclosing class. In C the name of the nested class belongs to the same scope as the name of the outermost enclosing class.

Example:

```
struct X {
        struct Y { /* ... */ } y;
};
struct Y yy;                    // valid C, invalid C++
```

Rationale: C++ classes have member functions which require that classes establish scopes. The C rule would leave classes as an incomplete scope mechanism which would prevent C++ programmers from maintaining locality within a class. A coherent set of scope rules for C++ based on the C rule would be very complicated and C++ programmers would be unable to predict reliably the meanings of nontrivial examples involving nested or local functions.
Effect on original feature: Change of semantics of well-defined feature.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;                       // struct Y and struct X are at the same scope
struct X {
        struct Y { /* ... */ } y;
};
```

1 All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 3.3.
How widely used: Seldom.

9.9
Change: In C++, a typedef name may not be redeclared in a class definition after being used in that definition

Example:

```
typedef int I;
struct S {
        I i;
        int I;                  // valid C, invalid C++
};
```

Rationale: When classes become complicated, allowing such a redefinition after the type has been used can create confusion for C++ programmers as to what the meaning of 'I' really is.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. Either the type or the struct member has to be renamed.
How widely used: Seldom.

C.1.8 Clause 12: special member functions [diff.special]

12.8 (copying class objects)
Change: Copying volatile objects

1 The implicitly-declared copy constructor and implicitly-declared copy assignment operator cannot make a copy of a volatile lvalue. For example, the following is valid in ISO C:

```
struct X { int i; };
struct X x1, x2;
volatile struct X x3 = {0};
x1 = x3;                        // invalid C++
x2 = x3;                        // also invalid C++
```

Rationale: Several alternatives were debated at length. Changing the parameter to `volatile const X&` would greatly complicate the generation of efficient code for class objects. Discussion of providing two alternative signatures for these implicitly-defined operations raised unanswered concerns about creating ambiguities and complicating the rules that specify the formation of these operators according to the bases and members.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Semantic transformation. If volatile semantics are required for the copy, a user-declared constructor or assignment must be provided. [*Note:* this user-declared constructor may be explicitly defaulted. — *end note*] If non-volatile semantics are required, an explicit `const_cast` can be used.
How widely used: Seldom.

C.1.9 Clause 16: preprocessing directives [diff.cpp]

16.8 (predefined names)
Change: Whether `__STDC__` is defined and if so, what its value is, are implementation-defined
Rationale: C++ is not identical to ISO C. Mandating that `__STDC__` be defined would require that translators make an incorrect claim. Each implementation must choose the behavior that will be most useful to its marketplace.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Programs and headers that reference `__STDC__` are quite common.

C.2 Standard C library [diff.library]

1 This subclause summarizes the contents of the C++ standard library included from the Standard C library. It also summarizes the explicit changes in definitions, declarations, or behavior from the ISO/IEC 9899:1990 and ISO/IEC 9899:1990/DAM 1 noted in other subclauses (17.6.2.3, 18.1, 21.5).

2 The C++ standard library provides 56 standard macros from the C library, as shown in Table 125.

3 The header names (enclosed in < and >) indicate that the macro may be defined in more than one header. All such definitions are equivalent (3.2).

Table 125 — Standard macros

assert	HUGE_VAL	NULL <cstdlib>	SIG_ERR	TMP_MAX
BUFSIZ	L_tmpnam	NULL <cstring>	SIG_IGN	va_arg
CLOCKS_PER_SEC	LC_ALL	NULL <ctime>	SIGABRT	va_end
EDOM	LC_COLLATE	NULL <cwchar>	SIGFPE	va_start
EILSEQ	LC_CTYPE	offsetof	SIGILL	WCHAR_MAX
EOF	LC_MONETARY	RAND_MAX	SIGINT	WCHAR_MIN
ERANGE	LC_NUMERIC	SEEK_CUR	SIGSEGV	WEOF <cwchar>
errno	LC_TIME	SEEK_END	SIGTERM	WEOF <cwctype>
EXIT_FAILURE	MB_CUR_MAX	SEEK_SET	stderr	_IOFBF
EXIT_SUCCESS	NULL <clocale>	setjmp	stdin	_IOLBF
FILENAME_MAX	NULL <cstddef>	SIG_DFL	stdout	_IONBF
FOPEN_MAX				

4 The C++ standard library provides 57 standard values from the C library, as shown in Table 126.

Table 126 — Standard values

CHAR_BIT	FLT_DIG	INT_MIN	MB_LEN_MAX
CHAR_MAX	FLT_EPSILON	LDBL_DIG	SCHAR_MAX
CHAR_MIN	FLT_MANT_DIG	LDBL_EPSILON	SCHAR_MIN
DBL_DIG	FLT_MAX	LDBL_MANT_DIG	SHRT_MAX
DBL_EPSILON	FLT_MAX_10_EXP	LDBL_MAX	SHRT_MIN
DBL_MANT_DIG	FLT_MAX_EXP	LDBL_MAX_10_EXP	UCHAR_MAX
DBL_MAX	FLT_MIN	LDBL_MAX_EXP	UINT_MAX
DBL_MAX_10_EXP	FLT_MIN_10_EXP	LDBL_MIN	ULONG_MAX
DBL_MAX_EXP	FLT_MIN_EXP	LDBL_MIN_10_EXP	USRT_MAX
DBL_MIN	FLT_RADIX	LDBL_MIN_EXP	
DBL_MIN_10_EXP	FLT_ROUNDS	LONG_MAX	
DBL_MIN_EXP	INT_MAX	LONG_MIN	

5 The C++ standard library provides 20 standard types from the C library, as shown in Table 127.

Table 127 — Standard types

clock_t	ldiv_t	size_t <cstdio>	va_list
div_t	mbstate_t	size_t <cstdlib>	wctrans_t
FILE	ptrdiff_t	size_t <cstring>	wctype_t
fpos_t	sig_atomic_t	size_t <ctime>	wint_t <cwchar>
jmp_buf	size_t <cstddef>	time_t	wint_t <cwctype>

6 The C++ standard library provides 2 standard `struct`s from the C library, as shown in Table 128.

Table 128 — Standard structs

lconv	tm

7 The C++ standard library provides 209 standard functions from the C library, as shown in Table 129.

Table 129 — Standard functions

abort	fmod	isupper	mktime	strftime	wcrtomb
abs	fopen	iswalnum	modf	strlen	wcscat
acos	fprintf	iswalpha	perror	strncat	wcschr
asctime	fputc	iswcntrl	pow	strncmp	wcscmp
asin	fputs	iswctype	printf	strncpy	wcscoll
atan	fputwc	iswdigit	putc	strpbrk	wcscpy
atan2	fputws	iswgraph	putchar	strrchr	wcscspn
atexit	fread	iswlower	puts	strspn	wcsftime
atof	free	iswprint	putwc	strstr	wcslen
atoi	freopen	iswpunct	putwchar	strtod	wcsncat
atol	frexp	iswspace	qsort	strtok	wcsncmp
bsearch	fscanf	iswupper	raise	strtol	wcsncpy
btowc	fseek	iswxdigit	rand	strtoul	wcspbrk
calloc	fsetpos	isxdigit	realloc	strxfrm	wcsrchr
ceil	ftell	labs	remove	swprintf	wcsrtombs
clearerr	fwide	ldexp	rename	swscanf	wcsspn
clock	fwprintf	ldiv	rewind	system	wcsstr
cos	fwrite	localeconv	scanf	tan	wcstod
cosh	fwscanf	localtime	setbuf	tanh	wcstok
ctime	getc	log	setlocale	time	wcstol
difftime	getchar	log10	setvbuf	tmpfile	wcstombs
div	getenv	longjmp	signal	tmpnam	wcstoul
exit	gets	malloc	sin	tolower	wcsxfrm
exp	getwc	mblen	sinh	toupper	wctob
fabs	getwchar	mbrlen	sprintf	towctrans	wctomb
fclose	gmtime	mbrtowc	sqrt	towlower	wctrans
feof	isalnum	mbsinit	srand	towupper	wctype
ferror	isalpha	mbsrtowcs	sscanf	ungetc	wmemchr
fflush	iscntrl	mbstowcs	strcat	ungetwc	wmemcmp
fgetc	isdigit	mbtowc	strchr	vfprintf	wmemcpy
fgetpos	isgraph	memchr	strcmp	vfwprintf	wmemmove
fgets	islower	memcmp	strcoll	vprintf	wmemset
fgetwc	isprint	memcpy	strcpy	vsprintf	wprintf
fgetws	ispunct	memmove	strcspn	vswprintf	wscanf
floor	isspace	memset	strerror	vwprintf	

C.2.1 Modifications to headers [diff.mods.to.headers]

1 For compatibility with the Standard C library, the C++ standard library provides the 18 *C headers* (D.5), but their use is deprecated in C++.

C.2.2 Modifications to definitions [diff.mods.to.definitions]

C.2.2.1 Types `char16_t` and `char32_t` [diff.char16]

1 The types `char16_t` and `char32_t` are distinct types rather than typedefs to existing integral types.

C.2.2.2 Type `wchar_t` [diff.wchar.t]

1 `wchar_t` is a keyword in this International Standard (2.11). It does not appear as a type name defined in any of `<cstddef>`, `<cstdlib>`, or `<cwchar>` (21.5).

C.2.2.3 Header `<iso646.h>` [diff.header.iso646.h]

1 The tokens `and`, `and_eq`, `bitand`, `bitor`, `compl`, `not_eq`, `not`, `or`, `or_eq`, `xor`, and `xor_eq` are keywords in this International Standard (2.11). They do not appear as macro names defined in `<ciso646>`.

C.2.2.4 Macro `NULL` [diff.null]

1 The macro `NULL`, defined in any of `<clocale>`, `<cstddef>`, `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, or `<cwchar>`, is an implementation-defined C++ null pointer constant in this International Standard (18.1).

C.2.3 Modifications to declarations [diff.mods.to.declarations]

1 Header `<cstring>`: The following functions have different declarations:

- `strchr`
- `strpbrk`
- `strrchr`
- `strstr`
- `memchr`

21.5 describes the changes.

C.2.4 Modifications to behavior [diff.mods.to.behavior]

1 Header `<cstdlib>`: The following functions have different behavior:

- `atexit`
- `exit`
- `abort`

18.4 describes the changes.

2 Header `<csetjmp>`: The following functions have different behavior:

- `longjmp`

18.9 describes the changes.

C.2.4.1 Macro `offsetof(type,member-designator)` [diff.offsetof]

1 The macro `offsetof`, defined in `<cstddef>`, accepts a restricted set of `type` arguments in this International Standard. 18.1 describes the change.

C.2.4.2 Memory allocation functions [diff.malloc]

1 The functions `calloc`, `malloc`, and `realloc` are restricted in this International Standard. 20.7.15 describes the changes.

Annex D (normative) Compatibility features [depr]

1 This Clause describes features of the C++ Standard that are specified for compatibility with existing implementations.

2 These are deprecated features, where *deprecated* is defined as: Normative for the current edition of the Standard, but not guaranteed to be part of the Standard in future revisions.

D.1 Increment operator with `bool` operand [depr.incr.bool]

1 The use of an operand of type `bool` with the `++` operator is deprecated (see 5.3.2 and 5.2.6).

D.2 static keyword [depr.static]

1 The use of the `static` keyword is deprecated when declaring objects in namespace scope (see 3.3.5).

D.3 Access declarations [depr.access.dcl]

1 Access declarations are deprecated (see 11.3).

D.4 Implicit conversion from const strings [depr.string]

1 The implicit conversion from const to non-const qualification for string literals (4.2) is deprecated.

D.5 C standard library headers [depr.c.headers]

1 For compatibility with the C standard library and the C Unicode TR, the C++ standard library provides the 25 *C headers*, as shown in Table 130.

Table 130 — C headers

`<assert.h>`	`<float.h>`	`<math.h>`	`<stddef.h>`	`<tgmath.h>`
`<complex.h>`	`<inttypes.h>`	`<setjmp.h>`	`<stdio.h>`	`<time.h>`
`<ctype.h>`	`<iso646.h>`	`<signal.h>`	`<stdint.h>`	`<uchar.h>`
`<errno.h>`	`<limits.h>`	`<stdarg.h>`	`<stdlib.h>`	`<wchar.h>`
`<fenv.h>`	`<locale.h>`	`<stdbool.h>`	`<string.h>`	`<wctype.h>`

2 Every C header, each of which has a name of the form `name.h`, behaves as if each name placed in the standard library namespace by the corresponding `cname` header is placed within the global namespace scope. It is unspecified whether these names are first declared or defined within namespace scope (3.3.5) of the namespace `std` and are then injected into the global namespace scope by explicit *using-declaration*s (7.3.3).

3 [*Example:* The header `<cstdlib>` assuredly provides its declarations and definitions within the namespace `std`. It may also provide these names within the global namespace. The header `<stdlib.h>` assuredly provides the same declarations and definitions within the global namespace, much as in the C Standard. It may also provide these names within the namespace `std`. — *end example*]

D.6 Old iostreams members [depr.ios.members]

1 The following member names are in addition to names specified in Clause 27:

```
namespace std {
  class ios_base {
  public:
    typedef T1 io_state;
    typedef T2 open_mode;
    typedef T3 seek_dir;
    typedef OFF_T streamoff;
    typedef POS_T streampos;
    // remainder unchanged
  };
}
```

2 The type `io_state` is a synonym for an integer type (indicated here as `T1`) that permits certain member functions to overload others on parameters of type `iostate` and provide the same behavior.

3 The type `open_mode` is a synonym for an integer type (indicated here as `T2`) that permits certain member functions to overload others on parameters of type `openmode` and provide the same behavior.

4 The type `seek_dir` is a synonym for an integer type (indicated here as `T3`) that permits certain member functions to overload others on parameters of type `seekdir` and provide the same behavior.

5 The type `streamoff` is an implementation-defined type that satisfies the requirements of type *OFF_T* (27.4.1).

6 The type `streampos` is an implementation-defined type that satisfies the requirements of type *POS_T* (27.2).

7 An implementation may provide the following additional member function, which has the effect of calling `sbumpc()` (27.5.2.2.3):

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    void stossc();
    // remainder unchanged
  };
}
```

8 An implementation may provide the following member functions that overload signatures specified in Clause 27:

```
namespace std {
  template<class charT, class traits> class basic_ios {
  public:
    void clear(io_state state);
    void setstate(io_state state);
    void exceptions(io_state);
    // remainder unchanged
  };

  class ios_base {
  public:
    // remainder unchanged
  };
```

```
  template<class charT, class traits = char_traits<charT> >
  class basic_streambuf {
  public:
    pos_type pubseekoff(off_type off, ios_base::seek_dir way,
              ios_base::open_mode which = ios_base::in | ios_base::out);
    pos_type pubseekpos(pos_type sp,
              ios_base::open_mode which);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_filebuf : public basic_streambuf<charT,traits> {
  public:
    basic_filebuf<charT,traits>* open
    (const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_ifstream : public basic_istream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };

  template <class charT, class traits = char_traits<charT> >
  class basic_ofstream : public basic_ostream<charT,traits> {
  public:
    void open(const char* s, ios_base::open_mode mode);
    // remainder unchanged
  };
}
```

9 The effects of these functions is to call the corresponding member function specified in Clause 27.

D.7 `char*` streams [depr.str.strstreams]

1 The header `<strstream>` defines three types that associate stream buffers with character array objects and assist reading and writing such objects.

D.7.1 Class `strstreambuf` [depr.strstreambuf]

```
namespace std {
  class strstreambuf : public basic_streambuf<char> {
  public:
    explicit strstreambuf(streamsize alsize_arg = 0);
    strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
    strstreambuf(char* gnext_arg, streamsize n, char* pbeg_arg = 0);
    strstreambuf(const char* gnext_arg, streamsize n);

    strstreambuf(signed char* gnext_arg, streamsize n,
                 signed char* pbeg_arg = 0);
    strstreambuf(const signed char* gnext_arg, streamsize n);
    strstreambuf(unsigned char* gnext_arg, streamsize n,
                 unsigned char* pbeg_arg = 0);
    strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

```
    virtual ~strstreambuf();

    void  freeze(bool freezefl = true);
    char* str();
    int   pcount();

  protected:
    virtual int_type overflow (int_type c = EOF);
    virtual int_type pbackfail(int_type c = EOF);
    virtual int_type underflow();
    virtual pos_type seekoff(off_type off, ios_base::seekdir way,
                             ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual pos_type seekpos(pos_type sp, ios_base::openmode which
                               = ios_base::in | ios_base::out);
    virtual streambuf* setbuf(char* s, streamsize n);

  private:
    // typedef T1 strstate;                                    exposition only
    // static const strstate allocated;                        exposition only
    // static const strstate constant;                         exposition only
    // static const strstate dynamic;                          exposition only
    // static const strstate frozen;                           exposition only
    // strstate strmode;                                       exposition only
    // streamsize alsize;                                      exposition only
    // void* (*palloc)(size_t);                                exposition only
    // void (*pfree)(void*);                                   exposition only
  };
}
```

1 The class `strstreambuf` associates the input sequence, and possibly the output sequence, with an object of some *character* array type, whose elements store arbitrary values. The array object has several attributes.

2 [*Note:* For the sake of exposition, these are represented as elements of a bitmask type (indicated here as `T1`) called `strstate`. The elements are:

— `allocated`, set when a dynamic array object has been allocated, and hence should be freed by the destructor for the `strstreambuf` object;

— `constant`, set when the array object has `const` elements, so the output sequence cannot be written;

— `dynamic`, set when the array object is allocated (or reallocated) as necessary to hold a character sequence that can change in length;

— `frozen`, set when the program has requested that the array object not be altered, reallocated, or freed.

— *end note*]

3 [*Note:* For the sake of exposition, the maintained data is presented here as:

— `strstate strmode`, the attributes of the array object associated with the `strstreambuf` object;

— `int alsize`, the suggested minimum size for a dynamic array object;

— `void* palloc)(size_t)`, points to the function to call to allocate a dynamic array object;

— `void` (*`pfree)(void*)`, points to the function to call to free a dynamic array object.

— *end note*]

4 Each object of class `strstreambuf` has a *seekable area*, delimited by the pointers `seeklow` and `seekhigh`. If `gnext` is a null pointer, the seekable area is undefined. Otherwise, `seeklow` equals `gbeg` and `seekhigh` is either `pend`, if `pend` is not a null pointer, or `gend`.

D.7.1.1 `strstreambuf` constructors [depr.strstreambuf.cons]

```
explicit strstreambuf(streamsize alsize_arg = 0);
```

1 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 131.

Table 131 — `strstreambuf(streamsize)` effects

Element	Value
`strmode`	`dynamic`
`alsize`	`alsize_arg`
`palloc`	a null pointer
`pfree`	a null pointer

```
strstreambuf(void* (*palloc_arg)(size_t), void (*pfree_arg)(void*));
```

2 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 132.

Table 132 — `strstreambuf(void* (*)(size_t), void (*)(void*))` effects

Element	Value
`strmode`	`dynamic`
`alsize`	an unspecified value
`palloc`	`palloc_arg`
`pfree`	`pfree_arg`

```
strstreambuf(char* gnext_arg, streamsize n, char *pbeg_arg = 0);
strstreambuf(signed char* gnext_arg, streamsize n,
             signed char *pbeg_arg = 0);
strstreambuf(unsigned char* gnext_arg, streamsize n,
             unsigned char *pbeg_arg = 0);
```

3 *Effects:* Constructs an object of class `strstreambuf`, initializing the base class with `streambuf()`. The postconditions of this function are indicated in Table 133.

Table 133 — `strstreambuf(charT*, streamsize, charT*)` effects

Element	Value
`strmode`	0
`alsize`	an unspecified value
`palloc`	a null pointer
`pfree`	a null pointer

4 `gnext_arg` shall point to the first element of an array object whose number of elements `N` is determined as follows:

— If `n > 0`, `N` is `n`.

— If `n == 0`, `N` is `std::strlen(gnext_arg)`.

— If `n < 0`, `N` is `INT_MAX`.[335]

If `pbeg_arg` is a null pointer, the function executes:

```
setg(gnext_arg, gnext_arg, gnext_arg + N);
```

Otherwise, the function executes:

```
setg(gnext_arg, gnext_arg, pbeg_arg);
setp(pbeg_arg,  pbeg_arg + N);
```

```
strstreambuf(const char* gnext_arg, streamsize n);
strstreambuf(const signed char* gnext_arg, streamsize n);
strstreambuf(const unsigned char* gnext_arg, streamsize n);
```

Effects: Behaves the same as `strstreambuf((char*)gnext_arg,n)`, except that the constructor also sets `constant` in `strmode`.

```
virtual ~strstreambuf();
```

Effects: Destroys an object of class `strstreambuf`. The function frees the dynamically allocated array object only if `strmode & allocated != 0` and `strmode & frozen == 0`. (D.7.1.3 describes how a dynamically allocated array object is freed.)

D.7.1.2 Member functions [depr.strstreambuf.members]

```
void freeze(bool freezefl = true);
```

Effects: If `strmode` & `dynamic` is non-zero, alters the freeze status of the dynamic array object as follows:

— If `freezefl` is `true`, the function sets `frozen` in `strmode`.

— Otherwise, it clears `frozen` in `strmode`.

```
char* str();
```

Effects: Calls `freeze()`, then returns the beginning pointer for the input sequence, `gbeg`.

Remarks: The return value can be a null pointer.

```
int pcount() const;
```

Effects: If the next pointer for the output sequence, `pnext`, is a null pointer, returns zero. Otherwise, returns the current effective length of the array object as the next pointer minus the beginning pointer for the output sequence, `pnext - pbeg`.

D.7.1.3 `strstreambuf` overridden virtual functions [depr.strstreambuf.virtuals]

```
int_type overflow(int_type c = EOF);
```

Effects: Appends the character designated by `c` to the output sequence, if possible, in one of two ways:

335) The function signature `strlen(const char*)` is declared in `<cstring>`. (21.5). The macro `INT_MAX` is defined in `<climits>` (18.2).

— If `c != EOF` and if either the output sequence has a write position available or the function makes a write position available (as described below), assigns `c` to `*pnext++`.

Returns `(unsigned char)c`.

— If `c == EOF`, there is no character to append.

Returns a value other than `EOF`.

Returns `EOF` to indicate failure.

Remarks: The function can alter the number of write positions available as a result of any call.

To make a write position available, the function reallocates (or initially allocates) an array object with a sufficient number of elements `n` to hold the current array object (if any), plus at least one additional write position. How many additional write positions are made available is otherwise unspecified.[336] If `palloc` is not a null pointer, the function calls `(*palloc)(n)` to allocate the new dynamic array object. Otherwise, it evaluates the expression `new charT[n]`. In either case, if the allocation fails, the function returns `EOF`. Otherwise, it sets `allocated` in `strmode`.

To free a previously existing dynamic array object whose first element address is `p`: If `pfree` is not a null pointer, the function calls `(*pfree)(p)`. Otherwise, it evaluates the expression `delete[] p`.

If `strmode & dynamic == 0`, or if `strmode & frozen != 0`, the function cannot extend the array (reallocate it with greater length) to make a write position available.

```
int_type pbackfail(int_type c = EOF);
```

Puts back the character designated by `c` to the input sequence, if possible, in one of three ways:

— If `c != EOF`, if the input sequence has a putback position available, and if `(char)c == gnext[-1]`, assigns `gnext - 1` to `gnext`.

Returns `c`.

— If `c != EOF`, if the input sequence has a putback position available, and if `strmode` & `constant` is zero, assigns `c` to `*--gnext`.

Returns `c`.

— If `c == EOF` and if the input sequence has a putback position available, assigns `gnext - 1` to `gnext`.

Returns a value other than `EOF`.

Returns `EOF` to indicate failure.

Remarks: If the function can succeed in more than one of these ways, it is unspecified which way is chosen. The function can alter the number of putback positions available as a result of any call.

```
int_type underflow();
```

Effects: Reads a character from the *input sequence*, if possible, without moving the stream position past it, as follows:

— If the input sequence has a read position available, the function signals success by returning `(unsigned char)*gnext`.

336) An implementation should consider `alsize` in making this decision.

— Otherwise, if the current write next pointer `pnext` is not a null pointer and is greater than the current read end pointer `gend`, makes a *read position* available by assigning to `gend` a value greater than `gnext` and no greater than `pnext`.

Returns `(unsigned char*)gnext`.

Returns `EOF` to indicate failure.

Remarks: The function can alter the number of read positions available as a result of any call.

```
pos_type seekoff(off_type off, seekdir way, openmode which = in | out);
```

Effects: Alters the stream position within one of the controlled sequences, if possible, as indicated in Table 134.

Table 134 — `seekoff` positioning

Conditions	Result
`(which & ios::in) != 0`	positions the input sequence
`(which & ios::out) != 0`	positions the output sequence
`(which & (ios::in | ios::out)) == (ios::in | ios::out))` and `way ==` either `ios::beg` or `ios::end`	positions both the input and the output sequences
Otherwise	the positioning operation fails.

For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` as indicated in Table 135.

Table 135 — `newoff` values

Condition	`newoff` Value
`way == ios::beg`	0
`way == ios::cur`	the next pointer minus the beginning pointer (`xnext - xbeg`).
`way == ios::end`	`seekhigh` minus the beginning pointer (`seekhigh - xbeg`).
If `(newoff + off) < (seeklow - xbeg)`, or `(seekhigh - xbeg) < (newoff + off)`	the positioning operation fails

Otherwise, the function assigns `xbeg` + `newoff` + `off` to the next pointer `xnext`.

Returns: `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
pos_type seekpos(pos_type sp, ios_base::openmode which
                 = ios_base::in | ios_base::out);
```

Effects: Alters the stream position within one of the controlled sequences, if possible, to correspond to the stream position stored in `sp` (as described below).

— If `(which & ios::in) != 0`, positions the input sequence.

— If `(which & ios::out) != 0`, positions the output sequence.

— If the function positions neither sequence, the positioning operation fails.

24 For a sequence to be positioned, if its next pointer is a null pointer, the positioning operation fails. Otherwise, the function determines `newoff` from `sp.offset()`:

— If `newoff` is an invalid stream position, has a negative value, or has a value greater than (`seekhigh - seeklow`), the positioning operation fails

— Otherwise, the function adds `newoff` to the beginning pointer `xbeg` and stores the result in the next pointer `xnext`.

25 *Returns:* `pos_type(newoff)`, constructed from the resultant offset `newoff` (of type `off_type`), that stores the resultant stream position, if possible. If the positioning operation fails, or if the constructed object cannot represent the resultant stream position, the return value is `pos_type(off_type(-1))`.

```
streambuf<char>* setbuf(char* s, streamsize n);
```

26 *Effects:* Implementation defined, except that `setbuf(0, 0)` has no effect.

D.7.2 Class `istrstream` [depr.istrstream]

```
namespace std {
  class istrstream : public basic_istream<char> {
  public:
    explicit istrstream(const char* s);
    explicit istrstream(char* s);
    istrstream(const char* s, streamsize n);
    istrstream(char* s, streamsize n);
    virtual ~istrstream();

    strstreambuf* rdbuf() const;
    char *str();
  private:
    // strstreambuf sb;                                    exposition only
  };
}
```

1 The class `istrstream` supports the reading of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `strstreambuf` object.

D.7.2.1 `istrstream` constructors [depr.istrstream.cons]

```
explicit istrstream(const char* s);
explicit istrstream(char* s);
```

1 *Effects:* Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,0))`. `s` shall designate the first element of an NTBS.

```
istrstream(const char* s, streamsize n);
```

2 *Effects:* Constructs an object of class `istrstream`, initializing the base class with `istream(&sb)` and initializing `sb` with `strstreambuf(s,n))`. `s` shall designate the first element of an array whose length is `n` elements, and `n` shall be greater than zero.

D.7.2.2 Member functions [depr.istrstream.members]

```
strstreambuf* rdbuf() const;
```

1 *Returns:* `const_cast<strstreambuf*>(&sb)`.

```
char* str();
```

2 *Returns:* `rdbuf()->str()`.

D.7.3 Class `ostrstream` [depr.ostrstream]

```
namespace std {
  class ostrstream : public basic_ostream<char> {
  public:
    ostrstream();
    ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
    virtual ~ostrstream();

    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    char* str();
    int pcount() const;
  private:
    // strstreambuf sb;                                    exposition only
  };
}
```

1 The class `ostrstream` supports the writing of objects of class `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as:

— `sb`, the `strstreambuf` object.

D.7.3.1 `ostrstream` constructors [depr.ostrstream.cons]

```
ostrstream();
```

1 *Effects:* Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)` and initializing `sb` with `strstreambuf())`.

```
ostrstream(char* s, int n, ios_base::openmode mode = ios_base::out);
```

2 *Effects:* Constructs an object of class `ostrstream`, initializing the base class with `ostream(&sb)`, and initializing `sb` with one of two constructors:

— If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements.

The constructor is `strstreambuf(s, n, s)`.

— If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an NTBS whose first element is designated by `s`. The constructor is `strstreambuf(s, n, s + std::strlen(s))`.[337]

D.7.3.2 Member functions [depr.ostrstream.members]

```
strstreambuf* rdbuf() const;
```

1 *Returns:* `(strstreambuf*)&sb` .

```
void freeze(bool freezefl = true);
```

2 *Effects:* Calls `rdbuf()->freeze(freezefl)`.

```
char* str();
```

3 *Returns:* `rdbuf()->str()`.

```
int pcount() const;
```

4 *Returns:* `rdbuf()->pcount()`.

D.7.4 Class `strstream` [depr.strstream]

```
namespace std {
  class strstream
    : public basic_iostream<char> {
  public:
    // Types
    typedef char                                    char_type;
    typedef typename char_traits<char>::int_type int_type;
    typedef typename char_traits<char>::pos_type pos_type;
    typedef typename char_traits<char>::off_type off_type;

    // constructors/destructor
    strstream();
    strstream(char* s, int n,
              ios_base::openmode mode = ios_base::in|ios_base::out);
    virtual ~strstream();

    // Members:
    strstreambuf* rdbuf() const;
    void freeze(bool freezefl = true);
    int pcount() const;
    char* str();

  private:
  // strstreambuf sb;                                  exposition only
  };
}
```

1 The class `strstream` supports reading and writing from objects of classs `strstreambuf`. It supplies a `strstreambuf` object to control the associated array object. For the sake of exposition, the maintained data is presented here as

337) The function signature `strlen(const char*)` is declared in `<cstring>` (21.5).

— `sb`, the `strstreambuf` object.

D.7.4.1 `strstream` constructors [depr.strstream.cons]

```
strstream();
```

1 *Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)`.

```
strstream(char* s, int n,
          ios_base::openmode mode = ios_base::in|ios_base::out);
```

2 *Effects:* Constructs an object of class `strstream`, initializing the base class with `iostream(&sb)` and initializing `sb` with one of the two constructors:

— If `(mode & app) == 0`, then `s` shall designate the first element of an array of `n` elements. The constructor is `strstreambuf(s,n,s)`.

— If `(mode & app) != 0`, then `s` shall designate the first element of an array of `n` elements that contains an NTBS whose first element is designated by `s`. The constructor is `strstreambuf(s,n,s + std::strlen(s))`.

D.7.4.2 `strstream` destructor [depr.strstream.dest]

```
virtual ~strstream()
```

1 *Effects:* Destroys an object of class `strstream`.

```
strstreambuf* rdbuf() const;
```

2 *Returns:* `&sb`.

D.7.4.3 `strstream` operations [depr.strstream.oper]

```
void freeze(bool freezefl = true);
```

1 *Effects:* Calls `rdbuf()->freeze(freezefl)`.

```
char* str();
```

2 *Returns:* `rdbuf()->str()`.

```
int pcount() const;
```

3 *Returns:* `rdbuf()->pcount()`.

D.8 Binders [depr.lib.binders]

The binders `binder1st`, `bind1st`, `binder2nd`, and `bind2nd` are deprecated. [*Note:* The function template `bind` (20.6.12) provides a better solution. — *end note*]

D.8.1 Class template `binder1st` [depr.lib.binder.1st]

```
template <class Fn>
class binder1st
  : public unary_function<typename Fn::second_argument_type,
                          typename Fn::result_type> {
protected:
```

```
  Fn                      op;
  typename Fn::first_argument_type value;
public:
  binder1st(const Fn& x,
            const typename Fn::first_argument_type& y);
  typename Fn::result_type
    operator()(const typename Fn::second_argument_type& x) const;
  typename Fn::result_type
    operator()(typename Fn::second_argument_type& x) const;
};
```

1 The constructor initializes `op` with `x` and `value` with `y`.

2 `operator()` returns `op(value,x)`.

D.8.2 `bind1st` [depr.lib.bind.1st]

```
template <class Fn, class T>
  binder1st<Fn> bind1st(const Fn& fn, const T& x);
```

1 *Returns:* `binder1st<Fn>(fn, typename Fn::first_argument_type(x))`.

D.8.3 Class template `binder2nd` [depr.lib.binder.2nd]

```
template <class Fn>
class binder2nd
  : public unary_function<typename Fn::first_argument_type,
                          typename Fn::result_type> {
protected:
  Fn                       op;
  typename Fn::second_argument_type value;
public:
  binder2nd(const Fn& x,
            const typename Fn::second_argument_type& y);
  typename Fn::result_type
    operator()(const typename Fn::first_argument_type& x) const;
  typename Fn::result_type
    operator()(typename Fn::first_argument_type& x) const;
};
```

1 The constructor initializes `op` with `x` and `value` with `y`.

2 `operator()` returns `op(x,value)`.

D.8.4 `bind2nd` [depr.lib.bind.2nd]

```
template <class Fn, class T>
  binder2nd<Fn> bind2nd(const Fn& op, const T& x);
```

1 *Returns:* `binder2nd<Fn>(op, typename Fn::second_argument_type(x))`.

2 [*Example:*

```
  find_if(v.begin(), v.end(), bind2nd(greater<int>(), 5));
```

finds the first integer in vector `v` greater than 5;

```
find_if(v.begin(), v.end(), bind1st(greater<int>(), 5));
```

finds the first integer in `v` less than 5. — *end example*]

D.9 `auto_ptr` [depr.auto.ptr]

The class template `auto_ptr` is deprecated. [*Note:* The class template `unique_ptr` (20.7.12) provides a better solution. — *end note*]

D.9.1 Class template `auto_ptr` [auto.ptr]

1 Template `auto_ptr` stores a pointer to an object obtained via `new` and deletes that object when it itself is destroyed (such as when leaving block scope 6.7).

2 Template `auto_ptr_ref` holds a reference to an `auto_ptr`. It is used by the `auto_ptr` conversions to allow `auto_ptr` objects to be passed to and returned from functions.

```
namespace std {
  template <class Y> struct auto_ptr_ref { };

  template <class X> class auto_ptr {
  public:
    typedef X element_type;

    // D.9.1.1 construct/copy/destroy:
    explicit auto_ptr(X* p =0) throw();
    auto_ptr(auto_ptr&) throw();
    template<class Y> auto_ptr(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr&) throw();
    template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
    auto_ptr& operator=(auto_ptr_ref<X> r) throw();
   ~auto_ptr() throw();

    // D.9.1.2 members:
    X& operator*() const throw();
    X* operator->() const throw();
    X* get() const throw();
    X* release() throw();
    void reset(X* p =0) throw();

    // D.9.1.3 conversions:
    auto_ptr(auto_ptr_ref<X>) throw();
    template<class Y> operator auto_ptr_ref<Y>() throw();
    template<class Y> operator auto_ptr<Y>() throw();
  };

  template <> class auto_ptr<void>
  {
  public:
    typedef void element_type;
  };
}
```

3 The `auto_ptr` provides a semantics of strict ownership. An `auto_ptr` owns the object it holds a pointer to. Copying an `auto_ptr` copies the pointer and transfers ownership to the destination. If more than one

`auto_ptr` owns the same object at the same time the behavior of the program is undefined. [*Note:* The uses of `auto_ptr` include providing temporary exception-safety for dynamically allocated memory, passing ownership of dynamically allocated memory to a function, and returning dynamically allocated memory from a function. `auto_ptr` does not meet the `CopyConstructible` and `Assignable` requirements for standard library container elements and thus instantiating a standard library container with an `auto_ptr` results in undefined behavior. — *end note*]

D.9.1.1 `auto_ptr` constructors [auto.ptr.cons]

```
explicit auto_ptr(X* p =0) throw();
```

1 *Postconditions:* `*this` holds the pointer `p`.

```
auto_ptr(auto_ptr& a) throw();
```

2 *Effects:* Calls `a.release()`.

3 *Postconditions:* `*this` holds the pointer returned from `a.release()`.

```
template<class Y> auto_ptr(auto_ptr<Y>& a) throw();
```

4 *Requires:* `Y*` can be implicitly converted to `X*`.

5 *Effects:* Calls `a.release()`.

6 *Postconditions:* `*this` holds the pointer returned from `a.release()`.

```
auto_ptr& operator=(auto_ptr& a) throw();
```

7 *Requires:* The expression `delete get()` is well formed.

8 *Effects:* `reset(a.release())`.

9 *Returns:* `*this`.

```
template<class Y> auto_ptr& operator=(auto_ptr<Y>& a) throw();
```

10 *Requires:* `Y*` can be implicitly converted to `X*`. The expression `delete get()` is well formed.

11 *Effects:* `reset(a.release())`.

12 *Returns:* `*this`.

```
~auto_ptr() throw();
```

13 *Requires:* The expression `delete get()` is well formed.

14 *Effects:* `delete get()`.

D.9.1.2 `auto_ptr` members [auto.ptr.members]

```
X& operator*() const throw();
```

1 *Requires:* `get() != 0`

2 *Returns:* `*get()`

```
X* operator->() const throw();
```

3 *Returns:* `get()`

```
X* get() const throw();
```

Returns: The pointer `*this` holds.

```
X* release() throw();
```

Returns: `get()`

Postcondition: `*this` holds the null pointer.

```
void reset(X* p=0) throw();
```

Effects: If `get() != p` then `delete get()`.

Postconditions: `*this` holds the pointer `p`.

D.9.1.3 `auto_ptr` conversions [auto.ptr.conv]

```
auto_ptr(auto_ptr_ref<X> r) throw();
```

Effects: Calls `p.release()` for the `auto_ptr p` that `r` holds.

Postconditions: `*this` holds the pointer returned from `release()`.

```
template<class Y> operator auto_ptr_ref<Y>() throw();
```

Returns: An `auto_ptr_ref<Y>` that holds `*this`.

```
template<class Y> operator auto_ptr<Y>() throw();
```

Effects: Calls `release()`.

Returns: An `auto_ptr<Y>` that holds the pointer returned from `release()`.

```
auto_ptr& operator=(auto_ptr_ref<X> r) throw()
```

Effects: Calls `reset(p.release())` for the `auto_ptr p` that `r` holds a reference to.

Returns: `*this`

D.10 Iterator primitives [depr.lib.iterator.primitives]

To simplify the use of iterators and provide backward compatibility with previous C++ Standard Libraries, the library provides several classes and functions.

The `iterator_traits` and supporting facilities described in this section are deprecated. [*Note:* the iterator concepts (24.1) provide the equivalent functionality using the concept mechanism. — *end note*]

D.10.1 Iterator traits [iterator.traits]

Iterator traits provide an auxiliary mechanism for accessing the associated types of an iterator. If `Iter` is the type of an iterator, the types

```
iterator_traits<Iter>::difference_type
iterator_traits<Iter>::value_type
iterator_traits<Iter>::iterator_category
```

shall be defined as the iterator's difference type, value type and iterator category (described below), respectively. In addition, the types

```
iterator_traits<Iter>::reference
iterator_traits<Iter>::pointer
```

shall be defined as the iterator's reference and pointer types, that is, for an iterator object `a`, the same type as the type of `*a` and `a->`, respectively. In the case of an output iterator, the types

```
iterator_traits<Iter>::difference_type
iterator_traits<Iter>::value_type
iterator_traits<Iter>::reference
iterator_traits<Iter>::pointer
```

may be defined as `void`.

2 The *category* of an iterator roughly describes which of the iterator concepts (24.1) the iterator satisfies. Iterator categories refer to iterators as defined by ISO/IEC 14882:2003, and can be one of *input iterator*, *output iterator*, *forward iterator*, *bidirectional iterator*, or *random access iterator*.

3 If the type `Iter` has nested types `difference_type`, `value_type`, `pointer`, `reference`, and `iterator_category`, then the template `iterator_traits<Iter>` is defined as

```
namespace std {
  template<class Iter> struct iterator_traits {
    typedef typename Iter::difference_type   difference_type;
    typedef typename Iter::value_type        value_type;
    typedef typename Iter::pointer           pointer;
    typedef typename Iter::reference         reference;
    typedef typename Iter::iterator_category iterator_category;
  };
}
```

otherwise, it is defined as

```
namespace std {
  template<class Iter> struct iterator_traits { };
}
```

4 For each iterator category, a partial specialization of the `iterator_traits` class template provides appropriate type definitions for programs that use the deprecated iterator traits mechanism. These partial specializations provide backward compatibility for unconstrained templates using iterators as specified by the corresponding requirements tables of ISO/IEC 14882:2003.

```
  concept IsReference<typename T> { } // exposition only
  template<typename T> concept_map IsReference<T&> { }

  concept IsPointer<typename T> { } // exposition only
  template<typename T> concept_map IsPointer<T*> { }

  template<Iterator Iter> struct iterator_traits<Iter> {
    typedef void                              difference_type;
    typedef void                              value_type;
    typedef void                              pointer;
    typedef void                              reference;
    typedef output_iterator_tag               iterator_category;
  };

  template<InputIterator Iter> struct iterator_traits<Iter> {
    typedef Iter::difference_type             difference_type;
    typedef Iter::value_type                  value_type;
    typedef Iter::pointer                     pointer;
    typedef Iter::reference                   reference;
```

```
  typedef input_iterator_tag               iterator_category;
};

template<ForwardIterator Iter>
  requires IsReference<Iter::reference> && IsPointer<Iter::pointer>
  struct iterator_traits<Iter> {
    typedef Iter::difference_type            difference_type;
    typedef Iter::value_type                 value_type;
    typedef Iter::pointer                    pointer;
    typedef Iter::reference                  reference;
    typedef forward_iterator_tag             iterator_category;
  };

template<BidirectionalIterator Iter>
  requires IsReference<Iter::reference> && IsPointer<Iter::pointer>
  struct iterator_traits<Iter> {
    typedef Iter::difference_type            difference_type;
    typedef Iter::value_type                 value_type;
    typedef Iter::pointer                    pointer;
    typedef Iter::reference                  reference;
    typedef bidirectional_iterator_tag       iterator_category;
  };

template<RandomAccessIterator Iter>
  requires IsReference<Iter::reference> && IsPointer<Iter::pointer>
  struct iterator_traits<Iter> {
    typedef Iter::difference_type            difference_type;
    typedef Iter::value_type                 value_type;
    typedef Iter::pointer                    pointer;
    typedef Iter::reference                  reference;
    typedef random_access_iterator_tag       iterator_category;
  };
```

— *end note*]

D.10.2 Basic iterator [iterator.basic]

1 The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```
namespace std {
  template<class Category, class T, class Distance = ptrdiff_t,
           class Pointer = T*, class Reference = T&>
  struct iterator {
        typedef T         value_type;
        typedef Distance  difference_type;
        typedef Pointer   pointer;
        typedef Reference reference;
        typedef Category  iterator_category;
  };
}
```

D.10.3 Standard iterator tags [std.iterator.tags]

1 The library introduces *category tag* classes which are used as compile time tags to distinguish the different iterator concepts when using the `iterator_traits` mechanism. They are: `input_iterator_tag`, `output_-`

`iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every iterator of type `Iter`, `iterator_traits<Iter>::iterator_category` shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace std {
  struct input_iterator_tag {};
  struct output_iterator_tag {};
  struct forward_iterator_tag: public input_iterator_tag {};
  struct bidirectional_iterator_tag: public forward_iterator_tag {};
  struct random_access_iterator_tag: public bidirectional_iterator_tag {};
}
```

D.10.4 Iterator backward compatibility [iterator.backward]

1 The library provides concept maps that allow iterators specified with `iterator_traits` to interoperate with algorithms that require iterator concepts. [*Example:*

```
struct random_iterator
{
  typedef std::input_iterator_tag iterator_category;
  typedef int                     value_type;
  typedef int                     difference_type;
  typedef int*                    pointer;
  typedef int                     reference;

  random_iterator(int remaining = 0) : remaining(remaining) { }

  int operator*() const { return std::rand(); }
  int* operator->() const { return 0; }

  random_iterator& operator++() { --remaining; return *this; }

  random_iterator operator++(int) {
    random_iterator tmp(*this); ++(*this); return tmp;
  }

  int remaining;

  friend bool
  operator==(const random_iterator& i, const random_iterator& j)
  {
    return i.remaining == j.remaining;
  }

  friend bool
  operator!=(const random_iterator& i, const random_iterator& j)
  {
    return i.remaining != j.remaining;
  }
};

void f(random_iterator i, random_iterator j) {
  std::copy(i, j, std::ostream_iterator<int>(std::cout, " ")); // OK: standard library produces concept
                                                               // map InputIterator<random_iterator>
}
```

— *end example*]

2 For all iterator types except output iterators, the associated types `difference_type`, `value_type`, `pointer` and `reference` are given the same values as their counterparts in `iterator_traits`. For output iterators, the `reference` type is deduced from the type of the output iterator's dereference operator.

3 When the `iterator_traits` specialization contains the nested types `difference_type`, `value_type`, `pointer`, `reference` and `iterator_category`, the `iterator_traits` specialization is considered to be *valid*.

[*Example:* The following example is well-formed. The backward-compatbility concept map for `InputIterator` does not match because `iterator_traits<int>` is not valid.

```
template<IntegralLike T> void f(T);
template<InputIterator T> void f(T);

void g(int x) {
  f(x); // okay
}
```

— *end example*]

4 The library shall provide a concept map `Iterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iter>::iterator_category` convertible to `output_iterator_tag`, and that meets the syntactic requirements of the `Iterator` concept.

5 The library shall provide a concept map `InputIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iter>::iterator_category` convertible to `input_iterator_tag`, and that meets the syntactic requirements of the `InputIterator` concept.

6 The library shall provide a concept map `ForwardIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iterator>::iterator_category` convertible to `forward_iterator_tag`, and that meets the syntactic requirements of the `ForwardIterator` concept.

7 The library shall provide a concept map `BidirectionalIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iterator>::iterator_category` convertible to `bidirectional_iterator_` and that meets the syntactic requirements of the `BidirectionalIterator` concept.

8 The library shall provide a concept map `RandomAccessIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iterator>::iterator_category` convertible to `random_access_iterator_` and that meets the syntactic requirements of the `RandomAccessIterator` concept.

Index