Concepts for the C++0x Standard Library: Utilities (Revision 2)

Douglas Gregor and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, lums}@osl.iu.edu

Document number: N2322=07-0182
Revises document number: N2082=06-0152
Date: 2007-06-24
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 20 of the C++ Standard Library in order to make full use of concepts [1]. Many of the changes in this document have been verified to work with ConceptGCC and its modified Standard Library implementation. We make every attempt to provide complete backward compatibility with the pre-concept Standard Library, and note each place where we have knowingly changed semantics.

This document is formatted in the same manner as the working draft of the C++ standard (N2284). Future versions of this document will track the working draft and the concepts proposal as they evolve. Wherever the numbering of a (sub)section matches a section of the working paper, the text in this document should be considered replacement text, unless editorial comments state otherwise. All editorial comments will have a gray background. Changes to the replacement text are categorized and typeset as additions, ~~removals~~, or ~~changes~~modifications..

Chapter 20 General utilities library [utilities]

2 The following clauses describe utility and allocator ~~requirements~~concepts, utility components, 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 ~~Requirements~~Concepts	`<concepts>`
20.2 Utility components	`<utility>`
20.3 Tuples	`<tuple>`
?? Type traits	`<type_traits>`
20.5 Function objects	`<functional>`

20.6 Memory	`<memory>`
	`<cstdlib>`
	`<cstring>`

?? Date and time	`<ctime>`

Replace the section [utility.requirements] with the following section [utility.concepts]

20.1 Concepts [utility.concepts]

1 The `<concepts>` header describes requirements on template arguments used throughout the C++ Standard Library.

Header `<concepts>` synopsis

```
namespace std {
  // 20.1.1, support concepts:
  concept SameType<typename T, typename U> {  }
  concept DerivedFrom<typename Derived, typename Base> { }
  concept ObjectType<typename T> { }

  // 20.1.2, comparisons:
  auto concept EqualityComparable<typename T, typename U = T> see below;
  auto concept LessThanComparable<typename T, typename U = T> see below;

  // 20.1.3, destruction:
  auto concept Destructible<typename T> see below;
```

```
// 20.1.4, construction:
auto concept Constructible<typename T, typename... Args> see below;
auto concept DefaultConstructible<typename T> see below;

// 20.1.5, copy and move:
auto concept MoveConstructible<typename T> see below;
auto concept CopyConstructible<typename T> see below;
auto concept MoveAssignable<typename T, typename U = T> see below;
auto concept CopyAssignable<typename T, typename U = T> see below;
auto concept Swappable<typename T> see below;

// 20.1.6, regular types:
auto concept Semiregular<typename T> see below;
auto concept Regular<typename T> see below;

// 20.1.7, convertibility:
auto concept Convertible<typename T, typename U> see below;

// 20.1.8, true:
concept True<bool> { }
concept_map True<true> { }

// 20.1.9, arithmetic concepts:
auto concept Addable<typename T, typename U = T> see below;
auto concept Subtractable<typename T, typename U = T> see below;
auto concept Multiplicable<typename T, typename U = T> see below;
auto concept Divisible<typename T, typename U = T> see below;
auto concept Modulus<typename T, typename U = T> see below;
auto concept Negatable<typename T> see below;
concept Arithmetic<typename T> see below;
concept Integral<typename T> see below;
concept SignedIntegral<typename T> see below;
concept UnsignedIntegral<typename T> see below;
concept Floating<typename T> see below;

// 20.1.10, logical concepts:
auto concept LogicalAnd<typename T, typename U = T> see below;
auto concept LogicalOr<typename T, typename U = T> see below;
auto concept LogicalNot<typename T> see below;

// 20.1.11, operator concepts:
auto concept Dereferenceable<typename T> see below;

// 20.1.12, callable:
auto concept Callable<typename F, typename... Args> see below;

// 20.1.13, predicates:
auto concept Predicate<typename F, typename... Args> see below;

// 20.1.14, allocators:
```

```
  concept Allocator<typename X> see below;
  concept AllocatorGenerator<typename X> see below;
  template<Allocator X> concept_map AllocatorGenerator<X> see below;
}
```

20.1.1 Support concepts [concept.support]

1 Concept SameType describes a same-type requirement ([temp.req]).

```
concept SameType<typename T, typename U> {  }
```

2 A program shall not provide a concept map for the SameType concept.

3 Concept DerivedFrom describes a derivation requirement ([temp.req]).

```
concept DerivedFrom<typename Derived, typename Base> { }
```

4 A program shall not provide a concept map for the DerivedFrom concept.

5 Concept ObjectType requires that the type T be an *object type* ([intro.object]).

```
concept ObjectType<typename T> { }
```

6 A program shall not provide a concept map for the ObjectType concept.

20.1.2 Comparisons [concept.comparison]

1 Concept EqualityComparable requires that two values be comparable with operator==.

```
auto concept EqualityComparable<typename T, typename U = T> {
  bool operator==(T a, U b);
  bool operator!=(T a, U b) { return !(a == b); }
}
```

2 When T and U are identical, operator== is an equivalence relation, that is, it has the following properties:

— For all a, a == a.

— If a == b, then b == a.

— If a == b and b == c, then a == c.

3 Concept LessThanComparable requires the ability to order values via operator<.

```
auto concept LessThanComparable<typename T, typename U = T> {
  bool operator<(T a, U b);
  bool operator>(U a, T b) { return b < a; }
  bool operator<=(U a, T b) { return !(b < a); }
  bool operator>=(T a, T b) { return !(a < b); }
}
```

4 operator< is a strict weak ordering relation (**??**)

20.1.3 Destruction [concept.destruct]

1 Concept `Destructible` requires a destructor.

```
auto concept Destructible<typename T> {
  T::~T();
}

T::~T();
```

2 *Postconditions:* All resources owned by the object are reclaimed.

3 *Throws:* nothing.

20.1.4 Construction [concept.construct]

1 Concept `Constructible` requires the existence of a constructor that accepts a given set of arguments.

```
auto concept Constructible<typename T, typename... Args>
  : Destructible<T>
{
  T::T(Args...);
}
```

2 Concept `DefaultConstructible` requires the existence of a default constructor.

```
auto concept DefaultConstructible<typename T> : Constructible<T> { }
```

20.1.5 Copy and move [concept.copymove]

1 Concept `MoveConstructible` requires the ability to construct an object from an rvalue of the same type, potentially altering the rvalue.

```
auto concept MoveConstructible<typename T>
  : Destructible<T>
{
  T::T(T&&);
}

T::T(T&& rv);
```

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*]

3 Concept `CopyConstructible` requires the ability to create a copy of an object.

```
auto concept CopyConstructible<typename T> : MoveConstructible<T> {
  T::T(const T&);

  axiom CopyPreservation(T x) {
    T(x) == x;
  }
}
```

4 Concept MoveAssignable requires the ability to assign to an object from an rvalue, potentially altering the rvalue.

```
auto concept MoveAssignable<typename T, typename U = T> {
  typename result_type;
  result_type operator=(T&, U&&);
}

result_type operator=(T&, U&& rv);
```

5 *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*]

6 Concept CopyAssignable requires the ability to assign to an object.

```
auto concept CopyAssignable<typename T, typename U = T> : MoveAssignable<T, U> {
  typename result_type;
  result_type operator=(T&, const U&);

  axiom CopyPreservation(T& x, U y) {
    (x = y, x) == y;
  }
}
```

The CopyAssignable requirements in N2284 specify that operator= must return a T&. This is too strong a requirement for most of the uses of CopyAssignable, so we have weakened CopyAssignable to not require anything of its return type. When we need a T&, we'll add that as an explicit requirement. See, e.g., the Integral concept.

7 Concept Swappable requires that two values can be swapped.

```
auto concept Swappable<typename T> {
  void swap(T&, T&);
}

void swap(T& t, T& u);
```

8 *Postconditions:* t has the value originally held by u, and u has the value originally held by t.

20.1.6 Regular types [concept.regular]

1 Concept Semiregular collects several common requirements supported by most C++ types.

```
auto concept Semiregular<typename T> : CopyConstructible<T>, CopyAssignable<T> {
  requires SameType<CopyAssignable<T>::result_type, T&>;
}
```

2 Concept Regular describes semi-regular types that have equality comparison operators.

```
auto concept Regular<typename T> : Semiregular<T>, DefaultConstructible<T>, EqualityComparable<T> { }
```

20.1.7 Convertibility [concept.convertible]

1 Concept Convertible requires an implicit conversion from one type to another.

```
auto concept Convertible<typename T, typename U> {
  operator U(T);
}
```

20.1.8 True [concept.true]

1 Concept `True` requires that its argument (a `bool` value that must be an integral constant expression) be true.

```
concept True<bool> { }
concept_map True<true> { }
```

2 A program shall not provide a concept map for the `True` concept.

20.1.9 Arithmetic concepts [concept.arithmetic]

1 Concept `Addable` requires an addition operator.

```
auto concept Addable<typename T, typename U = T> {
  typename result_type;
  result_type operator+(T, U);
}
```

2 Concept `Subtractable` requires a subtraction operator.

```
auto concept Subtractable<typename T, typename U = T> {
  typename result_type;
  result_type operator-(T, U);
}
```

3 Concept `Multiplicable` requires a multiplication operator.

```
auto concept Multiplicable<typename T, typename U = T> {
  typename result_type;
  result_type operator*(T, U);
}
```

4 Concept `Divisible` requires a division operator.

```
auto concept Divisible<typename T, typename U = T> {
  typename result_type;
  result_type operator/(T, U);
}
```

5 Concept `Modulus` requires a modulus operator.

```
auto concept Modulus<typename T, typename U = T> {
  typename result_type;
  result_type operator%(T, U);
}
```

6 Concept `Negatable` requires a negation operator.

```
auto concept Negatable<typename T> {
  typename result_type;
```

```
  result_type operator-(T);
}
```

7 Concept `Arithmic` requires all of the operations available on arithmetic types ([basic.fundamental]).

```
concept Arithmetic<typename T> : Regular<T>, LessThanComparable<T> {
  T::T(long long);

  T& operator++(T&);
  T operator++(T& t, int) { T tmp(t); ++t; return tmp; }
  T& operator--(T&);
  T operator--(T& t, int) { T tmp(t); --t; return tmp; }

  T operator+(T);
  T operator-(T);

  T operator*(T x, T y) { T z(x); return z *= y; }
  T operator/(T x, T y) { T z(x); return z /= y; }
  T operator+(T x, T y) { T z(x); return z += y; }
  T operator-(T x, T y) { T z(x); return z -= y; }

  T& operator*=(T&, T);
  T& operator/=(T&, T);
  T& operator+=(T&, T);
  T& operator-=(T&, T);

  requires CopyAssignable<T> && SameType<Assignable<T>::result_type, T&>;
}
```

8 Concept `Integral` describes the requirements for integral types.

```
concept Integral<typename T> : Arithmetic<T> {
  T operator~(T);

  T operator%(T x, T y)  { T z(x); return z %= y; }
  T operator&(T x, T y)  { T z(x); return z &= y; }
  T operator^(T x, T y)  { T z(x); return z ^= y; }
  T operator|(T x, T y)  { T z(x); return z |= y; }
  T operator<<(T x, T y) { T z(x); return z <<= y; }
  T operator>>(T x, T y) { T z(x); return z >>= y; }

  T& operator%=(T&, T);
  T& operator&=(T&, T);
  T& operator^=(T&, T);
  T& operator|=(T&, T);
  T& operator<<=(T&, T);
  T& operator>>=(T&, T);
}
```

9 Concept `SignedIntegral` describes the requirements for signed integral types.

```
concept SignedIntegral<typename T> : Integral<T> { }
```

10 For every signed integral type `T` ([basic.fundamental]), there exists an empty concept map `SignedIntegral<T>`.

11 Concept `UnsignedIntegral` describes unsigned integral types.

```
concept UnsignedIntegral<typename T> : Integral<T> { }
```

12 For every unsigned integral type `T` ([basic.fundamental]), there exists an empty concept map `UnsignedIntegral<T>`.

13 If `char` is a signed integral type, there shall exist an empty concept map `SignedIntegral<char>` otherwise, there shall exist an empty concept map `UnsignedIntegral<char>`.

14 If `wchar_t` is a signed integral type, there shall exist an empty concept map `SignedIntegral<wchar_t>` otherwise, there shall exist an empty concept map `UnsignedIntegral<wchar_t>`.

15 The `Floating` concept describes floating-point numbers.

```
concept Floating<typename T> : Arithmetic<T> { }
```

16 For every floating point type `T` ([basic.fundamental]), there exists an empty concept map `Floating<T>`.

20.1.10 Logical concepts [concept.logical]

1 Concept `LogicalAnd` requires an addition operator.

```
auto concept LogicalAnd<typename T, typename U = T> {
  bool operator&&(T, U);
}
```

2 Concept `LogicalOr` requires an addition operator.

```
auto concept LogicalOr<typename T, typename U = T> {
  bool operator||(T, U);
}
```

3 Concept `LogicalNot` requires an addition operator.

```
auto concept LogicalNot<typename T> {
  bool operator!(T);
}
```

20.1.11 Operator concepts [concept.operator]

1 Concept `Dereferenceable` requires a dereference operator `*`.

```
auto concept Dereferenceable<typename T> {
  typename reference;
  reference operator*(T);
}
```

20.1.12 Callable [concept.callable]

1 The `Callable` concepts requires that the given parameter `F` be callable given arguments of types `Args`.

```
auto concept Callable<typename F, typename... Args> {
  typename result_type;
  result_type operator()(F&, Args...);
}
```

20.1.13 Predicates [concept.predicate]

1 The `Predicate` concept requires that a function object be callable with some number of arguments, the result of which can be used in a context that requires a `bool`.

```
auto concept Predicate<typename F, typename... Args> : Callable<F, Args...> {
  requires Convertible<result_type, bool>;
}
```

2 Predicate function objects shall not apply any non-constant function through the predicate arguments.

20.1.14 Allocators [concept.allocator]

We have kept most of the text of [allocator.requirements] here, although much of it has been moved from tables into numbered paragraphs when translating the allocator requirements into concepts.

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 it. All of the containers (clause **??**) are parameterized in terms of allocators.

[[**Remove Table 39: Descriptive variable definitions**]]

[[**Remove Table 40: Allocator requirements**]]

2 ~~Table 40 describes the requirements on types manipulated through allocators.~~The `Allocator` concept describes the requirements on allocators. All the operations on the allocators are expected to be amortized constant time. ~~Table 33 describes the requirements on allocator types.~~

```
concept Allocator<typename X> : Regular<X> {
  ObjectType value_type               = typename X::value_type;
  MutableRandomAccessIterator pointer = typename X::pointer;
  RandomAccessIterator const_pointer  = typename X::const_pointer;
  typename reference                  = typename X::reference;
  typename const_reference            = typename X::const_reference;
  SignedIntegral difference_type      = typename X::difference_type;
  UnsignedIntegral size_type          = typename X::size_type;
  template<ObjectType T> class rebind = see below;

  requires Convertible<pointer, const_pointer> &&
           Convertible<pointer, void*> &&
           Convertible<pointer, value_type*> &&
           SameType<pointer::value_type, value_type> &&
           SameType<pointer::reference, value_type&> &&
           SameType<pointer::reference, reference>;

  requires Convertible<const_pointer, const void*> &&
```

```
            Convertible<const_pointer, const value_type&> &&
            SameType<const_pointer::value_type, value_type> &&
            SameType<const_pointer::reference, const value_type&> &&
            SameType<const_pointer::reference, const_reference>;

  requires SameType<rebind<value_type>, X>;

  pointer X::allocate(size_type n);
  pointer X::allocate(size_type n, const_pointer p);
  void X::deallocate(pointer p, size_type n);

  size_type X::max_size();

  template<ObjectType T>
    X::X(const rebind<T>& y);

  void X::construct(pointer p, const value_type&);
  template<typename V>
    requires Convertible<V, value_type>
    void X::construct(pointer p, V&&);

  void X::destroy(pointer p);

  pointer X::address(reference);
  const_pointer X::address(const_reference);
}

UnsignedIntegral size_type;
```

3 *Type:* a type that can represent the size of the largest object in the allocation model

```
SignedIntegral difference_type;
```

4 *Type:* a type that can represent the difference between any two pointers in the allocation model

```
template<ObjectType T> class rebind;
```

5 *Type:* The member classassociated template `rebind` in the table above is effectively a typedef templateis a template that produces allocators in the same family as `X`: if the name `Allocator`X is bound to `SomeAllocator<T>` `SomeAllocator<value_type>`, then `Allocator::rebind<U>::other`rebind<U> is the same type as `SomeAllocator<U>`. The resulting type `SameAllocator<U>` shall meet the requirements of the `Allocator` concept.

The default value for `rebind` is a template `R` for which `R<U>` is `X::template rebind<U>::other`.

The aforementioned default value for `rebind` can be implemented as follows:

```
template<typename Alloc>
struct rebind_allocator<typename Alloc> {
  template<typename U>
  using rebind = typename Alloc::template rebind<U>::other;
};
```

The default value for `rebind` in the `Allocator` concept is, therefore, `rebind_allocator<X>::template rebind`.

```
pointer X::allocate(size_type n);
pointer X::allocate(size_type n, const_pointer p);
```

6 *Effects:* Memory is allocated for `n` objects of type ~~`T`~~`value_type` but objects are not constructed. [1)]

7 *Returns:* ~~The result is a random access iterator.~~ A pointer to the allocated memory. [*Note:* If `n == 0`, the return value is unspecified. —*end note*]

8 *Throws:* `allocate` may raise an appropriate exception.

```
void X::deallocate(pointer p, size_type n);
```

9 *Preconditions:* All `n` ~~`T`~~`value_type` objects in the area pointed to by `p` shall be destroyed prior to this call. `n` shall match the value passed to `allocate` to obtain this memory. [*Note:* `p` shall not be ~~null~~singular. —*end note*]

Throws: Does not throw exceptions.

```
size_type X::max_size();
```

10 *Returns:* the largest value that can meaningfully be passed to `X::allocate()`

```
template<typename V>
  requires Constructible<value_type, V&&>
  void X::construct(pointer p, V&&);
```

The non-templated `X::construct` has been removed from the `Allocator` requirements because it implies that the `value_type` is `CopyConstructible` (which we do not want as a requirement in the `Allocator` concept). The templated version is more general, allowing in-place and move construction.

11 *Effects:* `::new((void*)p) T(forward<V>(v))`

```
void X::destroy(pointer p);
```

12 *Effects:* `((T*)p)->~T()`

13 The `AllocatorGenerator` concept describes the requirements on types that can be used to generate `Allocators`.

```
concept AllocatorGenerator<typename X> : Regular<X> {
  typename value_type               = typename X::value_type;
  template<typename T> class rebind = see below;
```

[1)] It is intended that `a.allocate` 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".

```
  requires SameType<rebind<value_type>, X>;
}

template<typename T> class rebind;
```

14 *Type:* The ~~member class~~associated template `rebind` ~~in the table above is effectively a typedef template~~is a template that produces allocator generators in the same family as `X`: if the name ~~`Allocator`~~`X` is bound to ~~`SomeAllocator<T>`~~ `SomeAllocator<value_type>`, then ~~`Allocator::rebind<U>::other`~~`rebind<U>` is the same type as `SomeAllocator<U>`. The default value for `rebind` is a template `R` for which `R<U>` is `X::template rebind<U>::other`.

15 Two allocators or allocator generators compare equal with == iff storage allocated from each can be deallocated via the other.

16 Every `Allocator` also meets the requirements of the `AllocatorGenerator` concept:

```
template<Allocator X>
concept_map AllocatorGenerator<X> {
  typedef Allocator<X>::value_type value_type;
  template<typename U> using rebind = Allocator<X>::rebind<U>;
}
```

17 Implementations of containers described in this International Standard are permitted to assume that their `Alloc~~ator~~` template parameter meets the following two additional requirements beyond those in ~~Table 40~~the Allocator concept.

— All instances of a given allocator type are required to be interchangeable and always compare equal to each other.

— ~~The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, std::size_t, and std::ptrdiff_t, respectively.~~ The requirements clause may contain the following additional requirements: `SameType<Alloc::pointer, Alloc::value_type*>`, `SameType<Alloc::const_pointer, const Alloc::value_type*>`, `SameType<Alloc::size_type, std::size_t>`, and `SameType<Alloc::difference_type, std::ptrdiff_t>`.

18 Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in ~~Table 40~~concept Allocator, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

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<~~class~~EqualityComparable T> bool operator!=(const T&, const T&);
    template<~~class~~LessThanComparable T> bool operator> (const T&, const T&);
    template<~~class~~LessThanComparable T> bool operator<=(const T&, const T&);
    template<~~class~~LessThanComparable T> bool operator>=(const T&, const T&);
  }
```

```
  // 20.2.2, forward/move:
  template <class T> struct identity;
  template <classObjectType T> T&& forward(typename identity<T>::type&&);
  template <classObjectType T> typename remove_reference<T>::type&& move(T&&);

  // 20.2.3, pairs:
  template <classObjectType T1,classObjectType T2> struct pair;
  template <classEqualityComparable T1, classEqualityComparable T2>
    bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
  template <classLessThanComparable T1, classLessThanComparable T2>
    bool operator< (const pair<T1,T2>&, const pair<T1,T2>&);
  template <classEqualityComparable T1, classEqualityComparable T2>
    bool operator!=(const pair<T1,T2>&, const pair<T1,T2>&);
  template <classLessThanComparable T1, classLessThanComparable T2>
    bool operator> (const pair<T1,T2>&, const pair<T1,T2>&);
  template <classLessThanComparable T1, classLessThanComparable T2>
    bool operator>=(const pair<T1,T2>&, const pair<T1,T2>&);
  template <classLessThanComparable T1, classLessThanComparable T2>
    bool operator<=(const pair<T1,T2>&, const pair<T1,T2>&);
  template <classSwappable T1, classSwappable T2>
    void swap(pair<T1,T2>&, pair<T1,T2>&);
  template <classSwappable T1, classSwappable T2>
    void swap(pair<T1,T2>&&, pair<T1,T2>&);
  template <classSwappable T1, classSwappable T2>
    void swap(pair<T1,T2>&, pair<T1,T2>&&);
  template <classCopyConstructible T1, classCopyConstructible T2> pair<T1,T2> make_pair(T1, T2);
}
```

20.2.1 Operators [operators]

By adding concept constraints to the operators in `rel_ops`, we eliminate nearly all of the problems with `rel_ops` that caused them to be banished. We could consider bringing them back into namespace `std`, if they are deemed useful.

1 To avoid redundant definitions of `operator!=` out of `operator==` and operators `>`, `<=`, and `>=` out of `operator<`, the library provides the following:

```
template <classEqualityComparable T> bool operator!=(const T& x, const T& y);
```

2 ~~Requires: Type T is EqualityComparable (20.1.1).~~

3 *Returns:* `!(x == y)`.

```
template <classLessThanComparable T> bool operator>(const T& x, const T& y);
```

4 ~~Requires: Type T is LessThanComparable (20.1.2).~~

5 *Returns:* `y < x`.

```
template <classLessThanComparable T> bool operator<=(const T& x, const T& y);
```

6 ~~Requires: Type T is LessThanComparable (20.1.2).~~

7 *Returns:* `!(y < x)`.

```
template <~~class~~LessThanComparable T> bool operator>=(const T& x, const T& y);
```

8 ~~Requires: Type T is LessThanComparable (20.1.2).~~

9 *Returns:* `!(x < y)`.

10 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 <class T> struct identity { typedef T type; };
```

2 [*Note:* The use of `identity` in `forward` forces users to explicitly specify the template parameter. This is necessary to get the correct forwarding semantics. — *end note*]

```
template <~~class~~ObjectType T> T&& forward(typename identity<T>::type&& t);
```

3 *Returns:* `t`.

```
template <~~class~~ObjectType T> typename remove_reference<T>::type&& move(T&& t);
```

7 *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 20.3.1.3 and 20.3.1.4).

```
template <~~class~~ObjectType T1, ~~class~~ObjectType 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, U&&> && Constructible<T2, V&&>
    pair(U&& x, V&& y);
  requires MoveConstructible<T1> && MoveConstructible<T2> pair(pair&& p);
  template<class U, class V>
    requires Constructible<T1, U> && Constructible<T2, V>
    pair(const pair<U, V>& p);
```

```
    template<class U, class V>
      requires Constructible<T1, U&&> && Constructible<T2, V&&>
      pair(pair<U, V>&& p);

    requires MoveAssignable<T1> && MoveAssignable<T2> pair& operator=(pair&& p );
    template<class U , class V>
      requires MoveAssignable<T1, U> && MoveAssignable<T2, V>
      pair& operator=(pair<U , V>&& p );

    requires Swappable<T1> && Swappable<T2> void swap(pair&& p );
  };
```

```
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, U&&> && Constructible<T2, V&&>
  pair(U&& x, V&& y);
```

4 *Effects:* The constructor initializes `first` with `forward<U>(`x`)` and `second` with `forward<T>(`y`)`.

```
requires MoveConstructible<T1> && MoveConstructible<T2> pair(pair&& p);
```

5 *Effects:* The constructor initializes `first` with `move(`p`.first)` and `second` with `move(`p`.second)`.

```
template<class U, class V>
  requires Constructible<T1, U> && Constructible<T2, V>
  pair(const pair<U, V> &p);
```

6 *Effects:* Initializes members from the corresponding members of the argument, performing implicit conversions as needed.

```
template<class U, class V>
  requires Constructible<T1, U&&> && Constructible<T2, V&&>
  pair(pair<U, V>&& p);
```

7 *Effects:* The constructor initializes `first` with `move(`p`.first)` and `second` with `move(`p`.second)`.

```
requires MoveAssignable<T1> && MoveAssignable<T2> pair& operator=(pair&& p);
```

8 *Effects:* Assigns to `first` with `move(`p`.first)` and to `second` with `move(`p`.second)`.

9 *Returns:* `*this`.

```
template<class U, class V>
  requires MoveAssignable<T1, U> && MoveAssignable<T2, V>
  pair& operator=(pair<U, V>&& p);
```

10 *Effects:* Assigns to `first` with `move(`p`.first)` and to `second` with `move(`p`.second)`.

11 *Returns:* `*this`.

```
requires Swappable<T1> && Swappable<T2> void swap(pair&& p);
```

12 *Effects:* Swaps `first` with p`.first` and `second` with p`.second`.

13 ~~*Requires:* `first_type` and `second_type` must be Swappable.~~

```
template <~~class~~EqualityComparable T1, ~~class~~EqualityComparable T2>
  bool operator==(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

14 *Returns:* `x.first == y.first && x.second == y.second`.

```
template <~~class~~LessThanComparable T1, ~~class~~LessThanComparable T2>
  bool operator<(const pair<T1, T2>& x, const pair<T1, T2>& y);
```

15 *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);
```

16 *Effects:* x`.swap(`y`)`

```
template <~~class~~CopyConstructible T1, ~~class~~CopyConstructible T2>
  pair<T1, T2> make_pair(T1 x, T2 y);
```

17 *Returns:* `pair<T1, T2>(x, y)`.[2)]

18 [*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*]

20.3 Tuples [tuple]

1 20.3 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**

[2)] According to (**??**), an implementation is permitted to not perform a copy of an argument, thus avoiding unnecessary copies.

```
namespace std {
  // 20.3.1, class template tuple:
  template <classObjectType... Types> class tuple;

  // 20.3.1.2, tuple creation functions:
  const unspecified ignore;

  template <class...  types>
    tuple<VTypes...> make_tuple(Types&&...);

  template<classObjectType... Types>
    tuple<Types&...> tie(Types&...);

  // 20.3.1.3, tuple helper classes:
  template <class T> class tuple_size;  // undefined
  template <class...  Types> class tuple_size<tuple<Types...> >;

  template <int I, class T> class tuple_element;    // undefined
  template <int I, class...  Types>
    requires True<0 <= I && I < sizeof...(Types)> class tuple_element<I, tuple<Types...> >;

  // 20.3.1.4, element access:
  template <int I, classObjectType... Types>
    requires True<0 <= I && I < sizeof...(Types)>
    typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>&);

  template <int I, classObjectType... Types>
    requires True<0 <= I && I < sizeof...(Types)>
    typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>&);

  // 20.3.1.5, 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...>&);

} // namespace std
```

20.3.1 Class template `tuple` [tuple.tuple]

1

2

```
template <classObjectType... Types>
class tuple
{
public:
  requires DefaultConstructible<Types>... tuple();
  requires CopyConstructible<Types>... explicit tuple(const Types&...);
  template <class...  UTypes>
    requires Constructible<Types, UTypes&&>...
    explicit tuple(UTypes&&...);

  requires CopyConstructible<Types>... tuple(const tuple&);
  requires MoveConstructible<Types>... tuple(tuple&&);

  template <class...  UTypes>
    requires Constructible<Types, UTypes>...
    tuple(const tuple<UTypes...>&);
  template <class...  UTypes>
    requires Constructible<Types, UTypes&&>...
    tuple(tuple<UTypes...>&&);

  template <class U1, class U2>
    tuple(const pair<U1, U2>&);      // iff sizeof...(Types) == 2
  template <class U1, class U2>
    tuple(pair<U1, U2>&&);           // iff sizeof...(Types) == 2

  requires CopyAssignable<Types>... tuple& operator=(const tuple&);
  requires MoveAssignable<Types>... tuple& operator=(tuple&&);

  template <class...  UTypes>
    requires CopyAssignable<Types, UTypes>...
    tuple& operator=(const tuple<UTypes...>&);
  template <class...  UTypes>
    requires MoveAssignable<Types, UTypes>...
    tuple& operator=(tuple<UTypes...>&&);

  template <class U1, class U2>
    tuple& operator=(const pair<U1, U2>&);  // iff sizeof...(Types) == 2
  template <class U1, class U2>
```

```
    tuple& operator=(pair<U1, U2>&&);       // iff sizeof...(Types) == 2
};
```

20.3.1.1 Construction [tuple.cnstr]

```
requires DefaultConstructible<Types>... tuple();
```

Requires: Each type in Types shall be default constructible.

Effects: Default initializes each element.

```
requires CopyConstructible<Types>... tuple(const Types&...);
```

Requires: Each type in Types shall be copy constructible.

Effects: Copy initializes each element with the value of the corresponding parameter.

```
template <class... UTypes>
  requires Constructible<Types, UTypes&&>...
  tuple(UTypes&&... u);
```

Requires: Each type in Types shall be move constructible from the corresponding type in UTypes. `sizeof...(Types) == sizeof...(UTypes)`.

Effects: Initializes the elements in the tuple with the corresponding value in `forward<UTypes>(u)`.

```
requires CopyConstructible<Types>... tuple(const tuple& u);
```

Requires: Each type in Types shall be copy constructible.

Effects: Copy constructs each element of `*this` with the corresponding element of `u`.

```
requires MoveConstructible<Types>... tuple(tuple&& u);
```

Requires: Each type in Types shall be move constructible.

Effects: Move-constructs each element of `*this` with the corresponding element of `u`.

```
template <class... UTypes>
  requires Constructible<Types, UTypes>...
  tuple(const tuple<UTypes...>& u);
```

Requires: Each type in Types shall be constructible from the corresponding type in UTypes. `sizeof...(Types) == sizeof...(UTypes)`.

Effects: Constructs each element of `*this` with the corresponding element of `u`.

[*Note:* `enable_if` can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — *end note*]

```
template <class... UTypes>
  requires Constructible<Types, UTypes&&>...
  tuple(tuple<UTypes...>&& u);
```

15 ~~*Requires:* Each type in Types shall be move-constructible from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).~~

16 *Effects:* Move-constructs each element of `*this` with the corresponding element of `u`.

[*Note:* `enable_if` can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. — *end note*]

```
template <class U1, class U2> tuple(const pair<U1, U2>& u);
```

17 *Requires:* The first type in `Types` shall be constructible from `U1` and the second type in `Types` shall be constructible from `U2`. `sizeof...(Types) == 2`.

18 *Effects:* Constructs the first element with `u.first` and the second element with `u.second`.

```
template <class U1, class U2> tuple(pair<U1, U2>&& u);
```

19 *Requires:* The first type in `Types` shall be move constructible from `U1` and the second type in `Types` shall be move-constructible from `U2`. `sizeof...(Types) == 2`.

20 *Effects:* Constructs the first element with `move(u.first)` and the second element with `move(u.second)`.

```
requires CopyAssignable<Types>... tuple& operator=(const tuple& u);
```

21 ~~*Requires:* Each type in Types shall be assignable.~~

22 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

23 *Returns:* `*this`

```
requires MoveAssignable<Types>... tuple& operator=(tuple&& u);
```

24 ~~*Requires:* Each type in Types shall be move-assignable.~~

25 *Effects:* Move-assigns each element of `u` to the corresponding element of `*this`.

26 *Returns:* `*this`.

```
template <class... UTypes>
  requires CopyAssignable<Types, UTypes>...
  tuple& operator=(const tuple<UTypes...>& u);
```

27 ~~*Requires:* Each type in Types shall be assignable from the corresponding type in UTypes~~.

28 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

29 *Returns:* `*this`

```
template <class... UTypes>
  requires MoveAssignable<Types, UTypes>...
  tuple& operator=(tuple<UTypes...>&& u);
```

30 ~~*Requires:* Each type in Types shall be move-assignable from the corresponding type in UTypes. sizeof...(Types) == sizeof...(UTypes).~~

31 *Effects:* Move-assigns each element of `u` to the corresponding element of `*this`.

32 *Returns:* `*this`.

```
template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);
```

33 *Requires:* The first type in `Types` shall be move assignable from `U1` and the second type in `Types` shall be move assignable from `U2`. `sizeof...(Types) == 2`.

34 *Effects:* Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

35 *Returns:* `*this`

36 [*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 U1, class U2> tuple& operator=(pair<U1, U2>&& u);
```

37 *Requires:* The first type in `Types` shall be assignable from `U1` and the second type in `Types` shall be assignable from `U2`. `sizeof...(Types) == 2`.

38 *Effects:* Assigns `move(u.first)` to the first element of `*this` and `move(u.second)` to the second element of `*this`.

39 *Returns:* `*this`.

20.3.1.2 Tuple creation functions [tuple.creation]

```
template<class... Types>
  tuple<VTypes...> make_tuple(Types&&... t);
```

1 where each `Vi` in `VTypes` is `X&` if, for the corresponding type `Ti` in `Types`, `remove_cv<remove_reference<Ti>::type>::type` equals `reference_wrapper<X>`, otherwise `Vi` is `decay<Ti>::type`.

2

3 *Returns:* `tuple<VTypes...>(forward<Types>(t)...)`.

4 [*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<~~class~~ObjectType... Types>
  tuple<Types&...> tie(Types&...  t);
```

Returns: `tuple<Types&>(t...)`. When an argument in `t` is `ignore`, assigning any value to the corresponding tuple element has no effect.

[*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*]

20.3.1.3 Tuple helper classes [tuple.helper]

```
template <int I, class...  Types>
requires True<0 <= I && I < sizeof...(Types)>
class tuple_element<I, tuple<Types...> > {
public:
  typedef TI type;
};
```

~~*Requires:* `0 <= I and I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.~~

Type: `TI` is the type of the `I`th element of `Types`, where indexing is zero-based.

20.3.1.4 Element access [tuple.elem]

```
template <int I, ~~class~~ObjectType... Types>
  requires True<0 <= I && I < sizeof...(Types)>
  typename tuple_element<I, tuple<Types...> >::type& get(tuple<Types...>& t);
```

~~*Requires:* `0 <= I and I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.~~

Returns: A reference to the `I`th element of `t`, where indexing is zero-based.

```
template <int I, ~~class~~ObjectType... Types>
  requires True<0 <= I && I < sizeof...(Types)>
  typename tuple_element<I, tuple<Types...> >::type const& get(const tuple<Types...>& t);
```

~~*Requires:* `0 <= I and I < sizeof...(Types)`. The program is ill-formed if `I` is out of bounds.~~

Returns: A const reference to the `I`th element of `t`, where indexing is zero-based.

7 [*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*]]

8 [*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*]

20.3.1.5 Relational operators [tuple.rel]

```
template<class... TTypes, class... UTypes>
  requires EqualityComparable<TTypes, UTypes>...
  bool operator==(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

1 ~~*Requires:* For all `i`, where `0 <= i and i < sizeof...(Types), get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes).`~~

2 *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`.

3 *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);
```

~~*Requires:* For all `i`, where `0 <= i and i < sizeof...(Types), get<i>(t) == get<i>(u)` is a valid expression returning a type that is convertible to `bool`. `sizeof...(TTypes) == sizeof...(UTypes).`~~

4 *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);
```

5 *Returns:* `!(t == u)`.

```
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<UTypes, TTypes>...
  bool operator<=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

7 *Returns:* `!(u < t)`

```
template<class... TTypes, class... UTypes>
  requires LessThanComparable<TTypes, UTypes>...
  bool operator>=(const tuple<TTypes...>& t, const tuple<UTypes...>& u);
```

8 *Returns:* `!(t < u)`

9 [*Note:* The above definitions for comparison operators do not require t_{tail} (or u_{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*]

20.5 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 **??**), 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 {
  // ??, base:
  template <class Arg, class Result> struct unary_function;
  template <class Arg1, class Arg2, class Result> struct binary_function;

  // ?? result_of:
  template <class> class result_of;    // undefined
  template <class F, class... Args> class result_of<F(ArgTypes...)>;

  // 20.5.5, reference_wrapper:
  template <class~~class~~ObjectType T> class reference_wrapper;

  template <class~~class~~ObjectType T> reference_wrapper<T> ref(T&);
  template <class~~class~~ObjectType T> reference_wrapper<const T> cref(const T&);

  template <class~~class~~ObjectType T> reference_wrapper<T> ref(reference_wrapper<T>);
  template <class~~class~~ObjectType T> reference_wrapper<const T> cref(reference_wrapper<T>);

  // 20.5.6, arithmetic operations:
  template <class T> struct plus;
  template <class T> struct minus;
  template <class T> struct multiplies;
  template <class T> struct divides;
  template <class T> struct modulus;
  template <class T> struct negate;

  // 20.5.7, comparisons:
  template <class T> struct equal_to;
  template <class T> struct not_equal_to;
  template <class T> struct greater;
  template <class T> struct less;
```

```
template <class T> struct greater_equal;
template <class T> struct less_equal;

// 20.5.8, logical operations:
template <class T> struct logical_and;
template <class T> struct logical_or;
template <class T> struct logical_not;

// ??, 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&);

// 20.5.10, bind:
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<classCopyConstructible Fn, classCopyConstructible... Types>
  unspecified bind(Fn, Types...);
template<classMoveConstructible R, classCopyConstructible Fn, classCopyConstructible... Types>
  unspecified bind(Fn, Types...);

namespace placeholders {
  // M is the implementation-defined number of placeholders
  extern unspecified _1;
  extern unspecified _2;
              .
              .
              .
  extern unspecified _M;
}

// ??, 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&);

// 20.5.11, adaptors:
template <classMoveConstructible Arg, classMoveConstructible Result>
  class pointer_to_unary_function;
template <classMoveConstructible Arg, classMoveConstructible Result>
  pointer_to_unary_function<Arg,Result> ptr_fun(Result (*)(Arg));
template <classMoveConstructible Arg1, classMoveConstructible Arg2, classMoveConstructible Result>
  class pointer_to_binary_function;
```

```
template <classMoveConstructible Arg1, classMoveConstructible Arg2, classMoveConstructible Result>
  pointer_to_binary_function<Arg1,Arg2,Result>
    ptr_fun(Result (*)(Arg1,Arg2));

// 20.5.12, adaptors:
template<classMoveConstructible S, class T> class mem_fun_t;
template<classMoveConstructible S, class T, classMoveConstructible A> class mem_fun1_t;
template<classMoveConstructible S, class T>
    mem_fun_t<S,T> mem_fun(S (T::*f)());
template<classMoveConstructible S, class T, classMoveConstructible A>
    mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A));
template<classMoveConstructible S, class T> class mem_fun_ref_t;
template<classMoveConstructible S, class T, classMoveConstructible A> class mem_fun1_ref_t;
template<classMoveConstructible S, class T>
    mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)());
template<classMoveConstructible S, class T, classMoveConstructible A>
    mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A));

template <classMoveConstructible S, class T> class const_mem_fun_t;
template <classMoveConstructible S, class T, classMoveConstructible A> class const_mem_fun1_t;
template <classMoveConstructible S, class T>
  const_mem_fun_t<S,T> mem_fun(S (T::*f)() const);
template <classMoveConstructible S, class T, classMoveConstructible A>
  const_mem_fun1_t<S,T,A> mem_fun(S (T::*f)(A) const);
template <classMoveConstructible S, class T> class const_mem_fun_ref_t;
template <classMoveConstructible S, class T, classMoveConstructible A> class const_mem_fun1_ref_t;
template <classMoveConstructible S, class T>
  const_mem_fun_ref_t<S,T> mem_fun_ref(S (T::*f)() const);
template <classMoveConstructible S, class T, classMoveConstructible A>
  const_mem_fun1_ref_t<S,T,A> mem_fun_ref(S (T::*f)(A) const);

// 20.5.13, member function adaptors:
template<classObjectType R, class T> unspecified mem_fn(R T::*);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::*)(Args...) pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::* const)(Args...) pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::* volatile)(Args...) pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::* const volatile)(Args...) pm);

// 20.5.14 polymorphic function wrappers:
class bad_function_call;

template<class> class function; // undefined
template<classMoveConstructible R, classMoveConstructible... ArgTypes>
  class function<R(ArgTypes...)>;

template<classMoveConstructible R, classMoveConstructible... ArgTypes>
```

```
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

  template<class R1, class R2, class... ArgTypes1, class... ArgTypes2>
    void operator==(const function<R1(ArgTypes1...)>&, const function<R2(ArgTypes2...)>&);
  template<class R1, class R2, class... ArgTypes1, class... ArgTypes2>
    void operator!=(const function<R1(ArgTypes1...)>&, const function<R2(ArgTypes2...)>&);

  template<~~class~~MoveConstructible R, ~~class~~MoveConstructible... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);
  template<~~class~~MoveConstructible R, ~~class~~MoveConstructible... ArgTypes>
    bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);
  template<~~class~~MoveConstructible R, ~~class~~MoveConstructible... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);
  template<~~class~~MoveConstructible R, ~~class~~MoveConstructible... ArgTypes>
    bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

  // ??, hash function base template:
  template <class 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<class T> struct hash<T*>;

  template <> struct hash<std::string>;
  template <> struct hash<std::u16string>;
  template <> struct hash<std::u32string>;
  template <> struct hash<std::wstring>;
}
```

20.5.5 Class template `reference_wrapper` [refwrap]

```
template <classObjectType 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;

  // tcode
  template <class...  ArgTypes>
    requires Callable<T, ArgTypes&&...>
    typename result_of<T(ArgTypes...)>::typeCallable<T, ArgTypes&&...>::result_type
  operator() (ArgTypes&&...) const;
};
```

20.5.5.4 `reference_wrapper` invocation [refwrap.invoke]

```
template <class...  ArgTypes>
  requires Callable<T, ArgTypes&&...>
  typename result_of<T(ArgTypes...)>::typeCallable<T, ArgTypes&&...>::result_type
    operator()(ArgTypes&&...  args) const;
```

1 *Returns:* *INVOKE*(`get()`, `forward<ArgTypes>(args)...`). (**??**)

2 *Remark:* `operator()` is described for exposition only. Implementations are not required to provide an actual `reference_wrapper::operator()`. Implementations are permitted to support `reference_wrapper` function invocation through multiple overloaded operators or through other means.

20.5.5.5 reference_wrapper helper functions [refwrap.helpers]

```
template <classObjectType T> reference_wrapper<T> ref(T& t);
```

1 *Returns:* `reference_wrapper<T>(t)`

2 *Throws:* nothing.

```
template <classObjectType T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

3 *Returns:* `ref(t.get())`

4 *Throws:* nothing.

```
template <classObjectType T> reference_wrapper<const T> cref(const T& t);
```

5 *Returns:* `reference_wrapper <const T>(t)`

6 *Throws:* nothing.

```
template <classObjectType T> reference_wrapper<const T> cref(reference_wrapper<T> t);
```

7 *Returns:* `cref(t.get());`

8 *Throws:* nothing.

20.5.6 Arithmetic operations [arithmetic.operations]

1 The library provides basic function object classes for all of the arithmetic operators in the language (**??**, **??**).

```
template <class T> struct plus : binary_function<T,T,T> {
  requires Addable<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns x `+` y.

```
template <class T> struct minus : binary_function<T,T,T> {
  requires Subtractable<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns x `-` y.

```
template <class T> struct multiplies : binary_function<T,T,T> {
  requires Multiplicable<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

4 `operator()` returns x `*` y.

```
template <class T> struct divides : binary_function<T,T,T> {
  requires Divisible<T> && Convertible<T::result_type, T>
  T operator()(const T& x, const T& y) const;
};
```

5 `operator()` returns x `/` y.

```
template <class T> struct modulus : binary_function<T,T,T> {
  requires Modulus<T> && Convertible<T::result_type, T>
    T operator()(const T& x, const T& y) const;
};
```

6 `operator()` returns x `%` y.

```
template <class T> struct negate : unary_function<T,T> {
  requires Negatable<T> && Convertible<T::result_type, T>
    T operator()(const T& x) const;
};
```

7 operator() returns −x.

20.5.7 Comparisons [comparisons]

1 The library provides basic function object classes for all of the comparison operators in the language (**??**, **??**).

```
template <class T> struct equal_to : binary_function<T,T,bool> {
  requires EqualityComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

2 operator() returns x == y.

```
template <class T> struct not_equal_to : binary_function<T,T,bool> {
  requires EqualityComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

3 operator() returns x != y.

```
template <class T> struct greater : binary_function<T,T,bool> {
  requires LessThanComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

4 operator() returns x > y.

```
template <class T> struct less : binary_function<T,T,bool> {
  requires LessThanComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

5 operator() returns x < y.

```
template <class T> struct greater_equal : binary_function<T,T,bool> {
  requires LessThanComparable<T>
    bool operator()(const T& x, const T& y) const;
};
```

6 operator() returns x >= y.

```
template <class T> struct less_equal : binary_function<T,T,bool> {
  requires LessThanComparable<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.

20.5.8 Logical operations [logical.operations]

1 The library provides basic function object classes for all of the logical operators in the language (**??**, **??**, **??**).

```
template <class T> struct logical_and : binary_function<T,T,bool> {
  requires LogicalAnd<T>
    bool operator()(const T& x, const T& y) const;
};
```

2 `operator()` returns *x* `&&` *y*.

```
template <class T> struct logical_or : binary_function<T,T,bool> {
  requires LogicalOr<T>
    bool operator()(const T& x, const T& y) const;
};
```

3 `operator()` returns *x* `||` *y*.

```
template <class T> struct logical_not : unary_function<T,bool> {
  requires LogicalNot<T>
    bool operator()(const T& x) const;
};
```

4 `operator()` returns `!`*x*.

20.5.10 Template function `bind` [bind]

1 The template function `bind` returns an object that binds a function object passed as an argument to additional arguments.

2 ~~Binders `bind1st` and `bind2nd` take a function object `fn` of two arguments and a value `x` and return a function object of one argument constructed out of `fn` with the first or second argument correspondingly bound to `x`.~~

20.5.10.1 Function object binders [func.bind]

20.5.10.1.3 Function template `bind` [func.bind.bind]

```
template<~~class~~CopyConstructible F, ~~class~~CopyConstructible...  BoundArgs>
  unspecified bind(F f, BoundArgs...  bound_args);
```

1 *Requires:* ~~F and each Ti in BoundArgs shall be CopyConstructible.~~ *INVOKE* `(f, w1, w2, ..., wN)` (**??**) 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 (**??**). The effect of `g(u1, u2, ..., uM)` shall be *INVOKE*`(f, v1, v2, ..., vN, ~~result_of~~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<classMoveConstructible R, classCopyConstructible F, classCopyConstructible...  BoundArgs>
  unspecified bind(F f, BoundArgs...  bound_args);
```

3 *Requires:* ~~F and each Ti in BoundArgs shall be CopyConstructible.~~ *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.

20.5.11 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 <classMoveConstructible Arg, classMoveConstructible 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 <classMoveConstructible Arg, classMoveConstructible Result>
  pointer_to_unary_function<Arg, Result> ptr_fun(Result (*f)(Arg));
```

3 *Returns:* `pointer_to_unary_function<Arg, Result>(`f`)`.

```
template <classMoveConstructible Arg1, classMoveConstructible Arg2, classMoveConstructible 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 <classMoveConstructible Arg1, classMoveConstructible Arg2, classMoveConstructible 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*]

20.5.12 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 20.5.11.

```
template <classMoveConstructible S, class 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 <classMoveConstructible S, class T, classMoveConstructible 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<classMoveConstructible S, class T> mem_fun_t<S,T>
   mem_fun(S (T::*f)());
template<classMoveConstructible S, class T, classMoveConstructible 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 <classMoveConstructible S, class 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 <classMoveConstructible S, class T, classMoveConstructible 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<classMoveConstructible S, class T> mem_fun_ref_t<S,T>
```

```
   mem_fun_ref(S (T::*f)());
template<classMoveConstructible S, class T, classMoveConstructible 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 <classMoveConstructible S, class 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 <classMoveConstructible S, class T, classMoveConstructible 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<cclassMoveConstructible S, class T> const_mem_fun_t<S,T>
   mem_fun(S (T::*f)() const);
template<classMoveConstructible S, class T, classMoveConstructible 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 <classMoveConstructible S, class 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 <classMoveConstructible S, class T, classMoveConstructible 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<~~class~~MoveConstructible S, class T> const_mem_fun_ref_t<S,T>
   mem_fun_ref(S (T::*f)() const);
template<~~class~~MoveConstructible S, class T, ~~class~~MoveConstructible 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).

20.5.13 Function template `mem_fn` [func.memfn]

```
template<~~class~~ObjectType R, class T> unspecified mem_fn(R T::* pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::*)(Args...) pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::* const)(Args...) pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::* volatile)(Args...) pm);
template<MoveConstructible R, class T, MoveConstructible... Args>
  unspecified mem_fn(R (T::* const volatile)(Args...) pm);
```

1 *Returns:* A simple call wrapper ([**??**]) `fn` such that the expression `fn(t, a2, ..., aN)` is equivalent to *INVOKE*`(pm, t, a2, ..., aN)` ([**??**]). `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.

5 ~~*Remarks:* Implementations may implement `mem_fn` as a set of overloaded function templates.~~

20.5.14 Polymorphic function wrappers [func.wrap]

20.5.14.2 Class template `function` [func.wrap.func]

```
namespace std {
  template<class> class function; // undefined

  template<~~class~~MoveConstructible R, ~~class~~MoveConstructible...  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;
```

```
// 20.5.14.2.1, construct/copy/destroy:
explicit function();
function(unspecified-null-pointer-type);
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&&);

function& operator=(const function&);
function& operator=(function&&);
function& operator=(unspecified-null-pointer-type);
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 CopyConstructible<F> && Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function& operator=(reference_wrapper<F>);

~function();

// ??, function modifiers:
void swap(function&);

// ??, function capacity:
operator unspecified-bool-type() const;

// ??, function invocation:
R operator()(ArgTypes...) const;

// 20.5.14.2.5, function target access:
const std::type_info& target_type() const;
template <typename T>
  requires Callable<F, ArgTypes...> && Convertible<Callable<F, ArgTypes...>::result_type, R>
  T* target();
template <typename T>
  requires Callable<F, ArgTypes...> && Convertible<Callable<F, ArgTypes...>::result_type, R>
  const T* target() const;
```

```
  private:
    // ??, undefined operators:
    template<class R2, class... ArgTypes2> bool operator==(const function<R2(ArgTypes2...)>&);
    template<class R2, class... ArgTypes2> bool operator!=(const function<R2(ArgTypes2...)>&);
  };

  // 20.5.14.2.7, Null pointer comparisons:
  template <class MoveConstructible R, class MoveConstructible... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);

  template <class MoveConstructible R, class MoveConstructible... ArgTypes>
    bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

  template <class MoveConstructible R, class MoveConstructible... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, unspecified-null-pointer-type);

  template <class MoveConstructible R, class MoveConstructible... ArgTypes>
    bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>&);

  // 20.5.14.2.8, specialized algorithms:
  template <class MoveConstructible R, class MoveConstructible... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);
} // namespace std
```

20.5.14.2.1 `function` construct/copy/destroy [func.wrap.func.con]

```
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 ~~*Requires:* `f` shall be callable for argument types `ArgTypes` and return type `R`.~~

9 *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`

10 Otherwise, `*this` targets a copy of `f` or `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.

11 *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.

```
template<class F>
```

```
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  operator=(F f);
```

19 *Effects:* `function(f).swap(*this);`

20 *Returns:* `*this`

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes..> &&
           Convertible<Callable<F, ArgTypes...>::result_type
  function& operator=(F&& f);
```

21 *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)`.

22 *Returns:* `*this`.

```
template<class F>
  requires CopyConstructible<F> && Callable<F, ArgTypes...> &&
           Convertible<Callable<F, ArgTypes...>::result_type, R>
  function& operator=(reference_wrapper<F> f);
```

23 *Effects:* `function(f).swap(*this);`

24 *Returns:* `*this`

25 *Throws:* nothing.

20.5.14.2.5 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<F, ArgTypes...> && Convertible<Callable<F, ArgTypes...>::result_type, R>
  T* target();
template<typename T>
  requires Callable<F, ArgTypes...> && Convertible<Callable<F, ArgTypes...>::result_type, R>
  const T* target() const;
```

3 *Requires:* `T` is a function object type that is Callable (20.5.14.2) for parameter types `ArgTypes` and return type `R`.

4 *Returns:* If `type() == typeid(T)`, a pointer to the stored function target; otherwise a null pointer.

5 *Throws:* nothing.

20.5.14.2.7 null pointer comparison operators [func.wrap.func.nullptr]

```
template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>& f, unspecified-null-pointer-type);

template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator==(unspecified-null-pointer-type, const function<R(ArgTypes...)>& f);
```

1 *Returns:* `!f`.

2 *Throws:* nothing.

```
template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>& f, unspecified-null-pointer-type);

template <classMoveConstructible R, classMoveConstructible... ArgTypes>
  bool operator!=(unspecified-null-pointer-type, const function<R(ArgTypes...)>& f);
```

3 *Returns:* `(bool) f`.

4 *Throws:* nothing.

20.5.14.2.8 specialized algorithms [func.wrap.func.alg]

```
template<classMoveConstructible R, classMoveConstructible... ArgTypes>
  void swap(function<R(ArgTypes...)>& f1, function<R(ArgTypes...)>& f2);
```

1 *Effects:* `f1.swap(f2);`

20.6 Memory [memory]

1 **Header** `<memory>` **synopsis**

```
namespace std {
  // 20.6.1, the default allocator:
  template <class T> class allocator; // undefined
  template <> class allocator<void>;
  template <class T> requires ObjectType<T> && Destructible<T>
    class allocator<T>;
  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();

  concept_map AllocatorGenerator<allocator<void> > { }
  template <class T> requires ObjectType<T> && Destructible<T>
    concept_map Allocator<allocator<T> > { }

  // ??, raw storage iterator:
  template <class OutputIterator, class T> class raw_storage_iterator;

  // ??, temporary buffers:
  template <class T>
```

```
  pair<T*,ptrdiff_t> get_temporary_buffer(ptrdiff_t n);
template <class T>
  void return_temporary_buffer(T* p);

// ??, specialized algorithms:
template <class InputIterator, class ForwardIterator>
  ForwardIterator
    uninitialized_copy(InputIterator first, InputIterator last,
                       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);




// ?? Class unique_ptr:
template <class X> class unique_ptr;
template <class X> class default_delete;

// ??, Class bad_weak_ptr:
class bad_weak_ptr;

// ??, Class template shared_ptr:
template<class T> class shared_ptr;

// ??, 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);

// ??, shared_ptr specialized algorithms:
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// ??, 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);

// ??, 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);
```

```
  // ??, shared_ptr get_deleter:
  template<class D, class T> D* get_deleter(shared_ptr<T> const& p);

  // ??, Class template weak_ptr:
  template<class T> class weak_ptr;

  // ??, weak_ptr comparison:
  template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

  // ??, weak_ptr specialized algorithms:
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);

  // ??, Class enable_shared_from_this:
  template<class T> class enable_shared_from_this;
}
```

20.6.1 The default allocator [default.allocator]

```
namespace std {
  template <class T> class allocator; // undefined

  // 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>
  requires ObjectType<T> && Destructible<T>
  class allocator<T> {
   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();

    requires CopyConstructible<T> void construct(pointer p, const T& val);
    template <class U> requires Constructible<T, U&&> void construct(pointer p, U&& val);
    void destroy(pointer p);
  };
}
```

1 Class template `allocator` meets the requirements of the `AllocatorGenerator` class when it is instantiated with a `Destructible` object type or with `void`. When instantiated with an object type, `allocator` meets the requirements of the `Allocator` concept.

```
concept_map AllocatorGenerator<allocator<void> > { }
template <class T> requires ObjectType<T> && Destructible<T>
  concept_map Allocator<allocator<T> > { }
```

20.6.1.1 `allocator` members [allocator.members]

`pointer address(reference x) const;`

1 *Returns:* `&x`.

`const_pointer address(const_reference x) const;`

2 *Returns:* `&x`.

`pointer allocate(size_type n, allocator<void>::const_pointer hint=0);`

3 *Remarks:* Uses `::operator new(std::size_t)` (**??**).

4 [*Note:* In a container member function, the address of an adjacent element is often a good choice to pass for this 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`.

6 *Remark:* the storage is obtained by calling `::operator new(std::size_t)`, 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*)` (**??**), 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.

```
requires CopyConstructible<T> void construct(pointer p, const_reference val);
```

12 *Effects:* `::new((void *)p) T(val)`

```
template <class U> requires Constructible<T, U&&> void construct(pointer p, U&& val);
```

13 *Effects:* `::new((void *)p) T(std::forward<U>(val))`

```
void destroy(pointer p);
```

14 *Effects:* `p->~T()`

20.6.1.2 `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`.

The author of this document ran out of time to complete it. Sorry!

Bibliography

[1] Douglas Gregor and Bjarne Stroustrup. Proposed wording for concepts (revision 1). Technical Report N2307=07-0167, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, July 2007.