Concepts for C++0x

Jeremy Siek[1] Douglas Gregor[1] Ronald Garcia[1] Jeremiah Willcock[1]
Jaakko Järvi[2] Andrew Lumsdaine[1]

1. Indiana University, Open Systems Lab, Bloomington, IN 47405
2. Texas A&M University, Computer Science, College Station, TX 77843

Document number: N1758=05-0018
Date: 2005-01-17
Project: Programming Language C++, Evolution Working Group
Reply-to: Jeremy Siek <jsiek at osl.iu.edu>

Contents

1 Overview

This proposal describes language extensions that provide direct language support for *concepts* in C++. Concepts are at the core of generic programming and are used to specify the abstractions that facilitate generic libraries such as the C++ standard library. Despite their importance, concepts are not explicitly supported in C++. Rather, they exist primarily as documentation (e.g., the requirements tables in the standard) in conjunction with a loose set of programming conventions. The extensions in this proposal allow concepts to be expressed directly in C++ and incorporate the features of current best practice in generic programming. First-class concepts will provide enhanced quality and usability of generic libraries and of generic programming in general.

The advantages of adding direct support for concepts to C++ include:

1. **Improved error messages** when using generic libraries. Error messages due to the incorrect use of function and class templates are notoriously poor. By expressing constraints on templates via concepts the situation can be improved: the compiler will issue a concise and clear diagnostic if the requirements of a template are not satisfied by the user.

2. **Improved type checking** for template definitions. Currently, compilers delay full type checking of template definitions until instantiation. This makes debugging more difficult for library authors, especially with respect to verifying whether the documented constraints provide enough functionality for the template definition to type check. With this proposal, the compiler immediately type checks some definitions, based just on the constraints (and of course the context of the definition) but without knowledge of any instantiations.

3. **Lowering barriers to entry** for programmers wishing to write generic libraries. Writing a generic library now requires a deep understanding of C++ and a plethora of template tricks—including type traits, tag dispatching, and SFINAE—that emulate basic generic programming constructs. This proposal replaces this grab bag of tricks with a single, coherent model for generic programming.

The proposals by Stroustrup and Dos Reis [SDR03a, Str03, SDR03b] explore the design space of concepts, and are the first concrete steps towards direct language support for concepts in standard C++. This proposal selects certain promising choices within that design space and develops them further. Section 2

provides a review of generic programming and introduces the terminology used throughout this proposal. Section 3 discusses the rationale behind the design of the proposed extension. Section 4 details the proposed language features while Section 5 discusses their impact on users, the standard library, and compiler vendors. Appendix A provides illustrative definitions for standard library concepts and Appendix B describes the template tricks currently used to implement generic programming that are superseded by this proposal.

2 Review of generic programming and its terminology

Generic programming is a methodology for creating highly reusable and efficient algorithms[1]. Maximum reusability is achieved by minimizing the number of requirements that an algorithm places on its parameters.

Within C++, generic algorithms are expressed as function templates. For instance, consider an implementation of the `accumulate()` algorithm from the standard library:

```
template<typename Iterator, typename T>
T accumulate(Iterator first, Iterator last, T init) {
  while (first != last) init = init + *first++;
  return init;
}
```

The requirements of this function can be discerned from its implementation: the type `Iterator` must support increment, deference, and comparison operations and the type `T` must support assignment and addition to the result of dereferencing the iterator. Related requirements are encapsulated into a *concept* to allow for more concise expression and to enable reuse of specifications. Thus, a concept is a set of requirements on one or more types. For instance, the standard library concept Input Iterator encompasses the requirements placed on the type parameter `Iterator`. These requirements include syntactic requirements (e.g., the `++` operation must be defined), semantic requirements (e.g., operator `*` returns the value at the current position), and performance requirements (e.g., increment must operate in constant time). We refer to syntactic requirements that mandate the definition of certain operators or functions, such as the increment or comparison operators, as *concept operations*. The standard library defines many concepts, such as Random Access Iterator, Sequence, Equality Comparable, and Assignable.

To use `accumulate()`, it must be instantiated with types that meet all of its requirements. For instance, `int*` may be used for the `Iterator` type, because it fulfills all of the requirements (dereference, increment, comparison). Similarly, `int` may be used for `T`, because it supports addition and assignment. When a type meets the requirements of a concept, we say that the type *models* the concept; or, equivalently, that the type is *a model of* the concept. A generic algorithm can be used when the supplied type arguments meet the algorithm's requirements, which are often expressed as concept requirements. Any given concept can potentially have an infinite number of models.

Different concepts often share requirements. When a concept B includes all of the requirements of a concept A, we say that B *refines* A. In that case, every model of B is also a model of A, but the reverse is not necessarily true. Thus, if an algorithm is specified to require a model of A, it can be supplied a model of B. For instance, the iterators over an `std::vector` model the concept Random Access Iterator, but they can still be passed to `accumulate()` because Random Access Iterator refines Input Iterator. Figure 1 illustrates the relationship between standard library iterator concepts. An edge B $\rightarrow$ A in the taxonomy indicates that B refines A.

Generic algorithms must often refer to the types of the arguments and results of concept operations (e.g., the return type of the `*` operation on iterators). These types are fully determined by, but not necessarily included among, the set of types that model a concept. We call them *associated types*, and we require a means of referring to them. As an example, consider an implementation of the standard library `advance()` function that moves an iterator:

```
template<typename BidirectionalIterator>
void advance(BidirectionalIterator& x, difference_type n) {
  while (n > difference_type(0)) { ++x; --n; }
  while (n < difference_type(0)) { --x; ++n; }
}
```

[1] Although we focus primarily on algorithms, these notions apply to generic data structures as well.

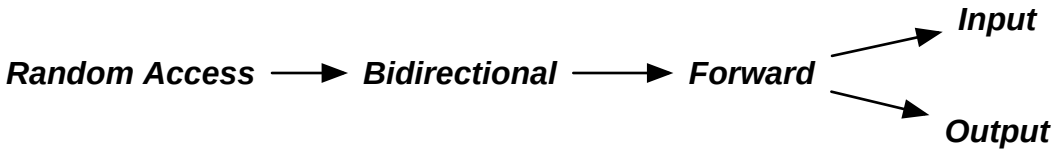

Figure 1: Iterator taxonomy

Here, *difference_type* is italicized to indicate that it should be replaced with the type used to measure distances between two iterators of type `BidirectionalIterator`. The difference type is implied by the particular model of Bidirectional Iterator: it may be `ptrdiff_t` for pointers or a 64-bit integer type for a file iterator. Within the standard library, the associated types of iterator concepts are accessible via the `iterator_traits` templates.

The above implementation of `advance()` requires a linear number of increments and decrements. However, when the type models Random Access Iterator the same operation can be performed in constant time using the `+=` operator, e.g.,

```
template<typename RandomAccessIterator>
void advance(RandomAccessIterator& x, difference_type n) {
  x += n;
}
```

Given a model of Random Access Iterator (say, a `vector` or `deque` iterator), either implementation can be correctly selected, because Random Access Iterator refines Bidirectional Iterator. However, the Random Access Iterator implementation is clearly better, because it takes constant time instead of linear time. Generic programming relies on the ability to select the *most specific* algorithm (i.e., the one that requires the most refined concepts that the incoming parameter types model), a feature we refer to as *concept-based function selection* (or, in the case of class templates, *concept-based partial ordering*).

Terminology review This section has described the fundamentals of generic programming and introduced some terminology. A summary of the key terms follows:

- A **concept** is a set of requirements on types.
- A concept **refines** another concept if it adds new requirements to the original concept.
- A set of types **model** (or are **a model of**) a concept if they meet its requirements.
- **Concept operations** are syntactic requirements that certain functions or operators be defined by a model.
- **Associated types** are part of a concept's specification and name the types that must be accessible via types that model the concept. They often denote parameter or return types of concept operations.
- **Concept-based function selection/partial ordering** involves selecting the most specific implementation of a component from a set of possibilities, based on the most refined concepts used in the component's specification.

3 Design rationale

We consider the following language features vital to support generic programming:

1. A way to define concepts, including associated types, concept operations, and concept refinement.
2. A syntax to explicitly declare how a set of types models a concept.

3. A way to express constraints on template parameters using concepts.
4. A way to order (partial) specializations and perform function selection based on the concept refinement relationships.

The features we enumerate above yield a large design space for generic programming facilities, much of which is explored in [SDR03a, Str03, SDR03b]. The following set of goals directed our extended evaluation of the viable design points and ultimately shaped the extensions we propose.

1. Earlier and more complete checking of template definitions.
2. Earlier and more complete checking of template uses.
3. Clearer and more helpful error messages.
4. Selection of template specialization/generic algorithm implementations based on attributes of template arguments.
5. Zero abstraction penalty.
6. Relatively simple to implement in existing compilers.
7. No C++03 programs become ill-formed.
8. Ablility to express a new, concept-aware standard library that is easier to use.
9. A simple migration path from the old standard library to the new.
10. A simple migration path for template library authors that want to add concepts.
11. Simple but powerful expression of constraints, including composition of constraints.

This section describes and motivates some of the key design decisions we reached in light of the above language requirements and goals. We discuss the compilation model for templates and the form of language features needed to support generic programming. We assume the reader is familiar with the concept papers by Stroustrup and Dos Reis [SDR03a, Str03, SDR03b] and only focus on the most promising of the design options.

3.1 Template compilation model

Adding concepts to C++ leaves the template compilation model largely unchanged. The language retains the instantiation model, where templates are in essence "patterns" that are stamped out for each combination of template parameters. Instantiations are still compiled in the same way, e.g., via the inclusion model or link-time instantiation, and the semantics of exported templates are likewise unchanged. Most importantly, existing templates will continue to work as expected, and can interact with the proposed extensions both at the language level and at the object code level.

This proposal introduces facilities for improved type checking of both template uses and definitions. The new type checking does not affect existing templates, and therefore cannot break backward compatibility. Rather, we introduce additional interface checking for templates and new "opaque" type parameters that bring with them additional type safety and eliminate many of the confusing aspects of templates.

Improved type checking for uses of templates solves one of the most frustrating aspects of using generic libraries in C++03: errors in the use of function templates are reported deep within the library implementation and typically refer to library implementation details. With the proposed extensions, a generic library can publish its requirements in the function interface, so that the compiler will check these requirements at the call site. Then, the user will receive an error message *at the call site* and it will refer to an interface violation, e.g., "the type `T` does not model the required concept `C`".

Improved type checking for template definitions also simplifies the task of writing correct function and class templates. Since a function template's interface expresses the constraints on its parameters, the compiler

can check that the definition does not require functionality beyond what is guaranteed by the constraints. For instance, if the constraints state that the input type `Iter` must model the Bidirectional Iterator concept but the definition uses the `+` operator, the compiler will produce an error message at template definition time. With existing C++03 templates, this error would go undetected until a user attempts to instantiate the template with a type that does not support `+`. For instance, the following function uses operations not provided by `InputIterator`, but the error will not be detected until it is instantiated with an iterator that does not provide `operator<`:

```
template<typename InputIterator, typename OutputIterator, typename Pred>
OutputIterator
copy_if(InputIterator first, InputIterator last, OutputIterator out, Pred pred) {
  while (first < last) {
    if (pred(*first)) *out++ = *first;
    ++first;
  }
  return out;
}
```

This proposal does *not* introduce support for separate compilation of templates to object code. The instantiation model of C++ templates is important for attaining efficient object code from generic code but does not translate well to separate compilation. The crucial aspect of the instantiation model is that the context of an instantiation affects the instantiation itself. The presence of template specializations, function overloads, and models means that two instantiations of a generic function can produce radically different code, such that no single compilation of that generic function will suffice.

3.2 Specific language features in the design space

The decisions above still leave a large design space. Here we illuminate some points of that space and justify our choices.

3.2.1 Named conformance vs. structural conformance

In general, there are two basic approaches to establishing that a type adheres to an interface: structural and named conformance. Structural conformance relies only on the signatures within an interface, ignoring the name of the interface itself. Thus, two interfaces with different names but the same content (or structure) are really the same interface. There are no declarations stating that a particular type implements an interface; instead this is checked when an object is passed to a function that requires the interface. In some sense, templates in C++03 use structural conformance: since the names of concepts do not matter, the template definition will compile (or not) solely based on whether certain functions exist for a given type.

Named conformance relies primarily on the name of an interface, so two interfaces with different names are distinct even if their structure is the same. With named conformance, an explicit declaration establishes that a type implements an interface. Subtyping in C++ uses named conformance and is established by an explicit declaration that one class inherits from another.

We advocate named conformance instead of structural conformance because it has clear advantages in implementability and offers stronger semantic guarantees. In the context of generic programming, named conformance means that an explicit *model declaration* is required to establish that a type models a particular concept. For instance, the declaration below establishes that the type `my_file_iterator` is a model of the Random Access Iterator concept.

```
model RandomAccessIterator<my_file_iterator> {};
```

With model declarations, the structure of types is verified when the model is defined (or instantiated): if the structure fails to meet the requirements, the compiler issues a diagnostic and may immediately abort compilation. After model declarations have been checked, it is not necessary to reverify that the structure of the type (i.e., all its operations) satisfies the requirements in the concept. In the case of structural conformance, the compiler may have to check the requirements of several concepts (e.g. to determine the best matching overload), many of which will fail. The compiler must back out of the failing checks without itself failing, essentially requiring the equivalent of an `is_compilable_expression` meta-function. Functionality

similar to `is_compilable_expression` has been requested before but was deemed problematic because it is considered difficult to implement.

Named conformance is safer than structural conformance in that a type cannot accidentally conform to a concept whose semantics are not met. For instance, the standard library Forward Iterator and Input Iterator concepts are almost syntactically identical. It is possible for a type to structurally match both Input Iterator and Forward Iterator, but for semantic reasons (e.g., it is only single pass) the type is not a Forward Iterator. If such a type is used with an algorithm that performs concept-based dispatching, the wrong implementation will be used. Named conformance requires the programmer to assert that the semantics meet the requirements thereby avoiding this kind of accidental conformance.

Structural conformance requires less typing for developers because explicit model declarations need not be written. This aids the transition from using current generic libraries to using concept-enabled libraries, because the user will not be required to write model declarations. However, there are several ways to ameliorate the inconvenience of explicit model declarations. First, an updated standard library will include model declarations for all the builtin types, standard classes, and concepts. For example, no user will need to write a model declaration to establish that `vector` is Equality Comparable when its value type is Equality Comparable. This will benefit users of the standard algorithms, as well as users of other algorithm libraries that use the standard concepts. Second, we can include model declarations in the standard library that apply to user-defined iterators (so long as they are standards conforming). Third, the C++ implementation will generate models for basic concepts such as Default Constructible and Copy Constructible for user-defined types. Finally, we envision that writing models declarations will become a just another part of defining user-defined types, and that code using these types will not need to be changed.

Another reason to prefer explicit model declarations is that they provide a simple mechanism for supporting associated types, which we discuss in Section 3.2.6. With model declarations, there is no need to use traits classes and partial specialization to provide access to associated types.

In summary, we prefer explicit model declarations over structural conformance because model declarations are straightfoward to implement in today's compilers, enable better semantic guarantees by preventing accidental conformance, and provide a programmer-friendly way to support associated types.

3.2.2 New templates vs. new template parameters

We have identified two possible approaches for introducing concept checking into C++ without breaking code that relies on the current "unchecked" template mechanism. First, we could leave the current templates unchanged and introduce a new entity, a fully-checked template. Alternatively, we could introduce a new kind of template type parameter, called an *opaque* parameter, that does not delay the checking of dependent expressions, as normal template parameters do. This second approach has several advantages. First, the fine-grained control will allow for easier migration to concept-enabled templates; the migration does not have to be all or nothing. Further, this approach allows for the definition of function templates whose uses are checked but whose body is not. This is important for templates that rely on type information from points of instantiation. For example, a type parameter may be used in the body of the template to instantiate another template, and the other template has partial or full specializations that must be chosen based on the concrete types of the template arguments.

We suggest the use of the `typeid` keyword to declare opaque template type parameters, whereas template type parameters introduced via the `typename` or `class` keyword retain their current behavior. When an expression depends only on concrete types and opaque template parameters, it can be type-checked at template definition time. The type arguments for the opaque parameters are unknown so argument-dependent lookup cannot be used to resolve operations involving these types. Instead, the template's constraints will introduce operations that may be applied to the opaque parameters.

A disadvantage of the opaque type parameter approach is that the presence of a single non-opaque template parameter could potentially eliminate all type checking for a template; users may not get the type safety that they think they have due to a relatively small change. However, templates that only use `typeid` will be fully checked.

```
template<typeid T>
concept LessThanComparable {
  bool operator<(T x, T y);
  bool operator>(T x, T y);
  bool operator<=(T x, T y);
  bool operator>=(T x, T y);
};
```

```
template<typeid T>
concept LessThanComparable {
  constraints(T x, T y) {
    bool b = e1 < e2;
    b = e1 > e2;
    b = e1 <= e2;
    b = e1 >= e2;
  }
};
```

Figure 2: Less Than Comparable concept expressed via pseudo-signatures (left) and valid expressions (right).

3.2.3 Pseudo-signatures vs. valid expressions

There are several ways to express the syntactic requirements of concepts. The two most feasible options are *pseudo-signatures* and *valid expressions* (sometimes called *usage patterns*). Stroustrup and Dos Reis [SDR03a] present other potential solutions and give solid reasons to discount all but these two. Both approaches are equivalent, in the sense that they can express the same constraints [JWL04]. However, certain types of constraints are easier to express in one approach or the other, and they require different levels of implementation complexity.

The pseudo-signature approach describes the syntax via a set of function declarations, as illustrated on the left side of Figure 2 (we describe the full syntax of concept definitions in Section 4.2). In a simple signatures approach, a type `T` would have to have functions that match those signatures *exactly*. A pseudo-signature approach, on the other hand, treats these declarations more loosely. For instance, the declaration of `operator<` requires the existence of a `<` operator, either built in, as a free function, or as a member function, that can be passed two values of type `T` and returns a value convertible to `bool`.

The valid-expression approach describes the valid syntax by writing it directly. The right side of Figure 2 illustrates the description of Less Than Comparable in a manner similar to that of Stroustrup [SDR03a]. The expressions that should be supported are written directly. A compiler can verify that a given type `T` conforms to this concept syntax by instantiating `constraints()` and reporting failure if there are any errors. This is the idea behind concept checking [Sie00, SL00], a C++ technique for simulating concept conformance checks to improve error messages.

Expressing syntactic constraints with valid expressions may lead to weaker constraints than intended. For example, the valid expression `bool b = e1 < e2` requires that the result of `e1 < e2` is convertible to `bool`, not that the result is of type `bool`. Such unintentionally weak constraints can lead to surprising behavior, as demonstrated by the following example. Consider the Less Than Comparable concept, whose specification uses the above valid expression formulation to state the requirement for `operator<`. With this specification, the following code would not type check:

```
template<typeid T> // require T to model LessThanComparable
bool foo(T x, T y) {
  return x < y && random() % 3;
}
```

The problem is that the type of `x < y`, even though convertible to `bool`, could have an overloaded `&&` operator taking an `int` on the right-hand side and returning some type not convertible to `bool`[2].

Our design for pseudo-signatures solves this problem. E.g., the requirement of in the Less Than Comparable concept would be written as `bool operator<(T, T)` stating explicitly that the result must be of type `bool`. From inside a constrained template, a pseudo-signature provides the exact declaration of a required operation. However, a pseudo-signature can be satisfied by functions with different signatures, as described in Section 4.2.1, and a conversion is performed by the C++ implementation.

The "fuzziness" of the valid expression approach also affects implementability. To check the body of a generic function against constraints specified as valid expressions, the compiler will have to construct a suitable representation for that. We believe this necessitates parsing all of the expressions and building a

[2]Standard library vendors have already had to deal with this problem, e.g., for `find`.

representation not unlike that of pseudo-signatures, possibly including several intermediate "convertible-to" types.

In some cases it is desirable to emulate a "fuzzy" requirement with pseudo-signatures, e.g., to state that an expression `e1 < e2` need only be convertible to `bool`. This can be expressed in pseudo-signatures by introducing additional associated types (described in Sections 3.2.6 and 4.2.2), and requiring convertibility explicitly. We can define the looser form of Less Than Comparable as follows (the `require` keyword is explained in Section 4.2.3):

```
template<typeid T>
concept LessThanComparable {
  typename compare_type;
  require Convertible<compare_type, bool>;

  compare_type operator<(T x, T y);
  compare_type operator>(T x, T y);
  compare_type operator<=(T x, T y);
  compare_type operator>=(T x, T y);
};
```

We have opted for pseudo-signatures because they ease compiler implementation and result in more precise concept specifications yet still provide sufficient flexibility.

3.2.4 Placeholder types vs. concept names as types

Several syntactic approaches to specifying concepts are possible. They could be written like a class, using the concept name for both the concept and for the type that models the concept, e.g.,

```
concept LessThanComparable {
  bool operator<(LessThanComparable x, LessThanComparable y);
  bool operator>(LessThanComparable x, LessThanComparable y);
  bool operator<=(LessThanComparable x, LessThanComparable y);
  bool operator>=(LessThanComparable x, LessThanComparable y);
};
```

However, this syntax is somewhat misleading because of its similarity to an abstract base class. If it were describing an abstract base class, then an object of any type that is Less Than Comparable could be compared to an object of another type that is Less Than Comparable, but this is not what the Less Than Comparable concept requires. We want to say that two objects of the *same* type can be compared, if that type models Less Than Comparable.

The requirements tables in the C++ standard use a different approach that introduces explicit placeholders for the types that model the concept. This approach can be applied to the definition of concepts by adopting syntax similar to that of templates. The template parameters are placeholders for the types that model the concept, e.g.,

```
template<typeid T>
concept LessThanComparable {
  bool operator<(T x, T y);
  bool operator>(T x, T y);
  bool operator<=(T x, T y);
  bool operator>=(T x, T y);
};
```

This syntax emphasizes that a concept is not a type and makes clear that operations are performed on specific models of a concept. With this syntax for Less Than Comparable, the requirements clearly express that the comparisons are between objects of type `T`, not between any two Less Than Comparable types.

This proposal advocates the placeholder approach for specifying concepts because it is the most widely-used approach in generic programming (and in the C++ standard) and because it better conveys the semantics of concepts. Additionally, some concepts place requirements on more than one type; the placeholder approach scales to such *multi-parameter concepts* without requiring new syntax.

3.2.5 Specifying constraints

The objective of adding concepts to C++ is to introduce checkable constraints for templates. These constraints may take many different forms, but we consider the two most promising options here: allowing concepts as the kind of a template parameter[3], and the use of `where` clauses.

One potential syntax for expressing constraints is to think of concepts as a "type of types" or a kind of template parameter. For instance, instead of using `typename` or `class` to introduce a template type parameter, one could provide the concept name. Consider the following function:

```
template<LessThanComparable T>
bool equivalent(const T& x, const T& y) {
  return !(x < y) && !(y < x);
}
```

The requirement that `T` model Less Than Comparable is expressed by making `LessThanComparable` the kind of the template type parameter `T`. This formulation has two weaknesses:

1. *Multi-parameter concepts* become hard to express. For instance, the concept Output Iterator has two parameters: the iterator type `Iter` and the value type `V`. The template parameter for an iterator type `Iter` would itself be expressed as a template, with nested parameters. For instance, consider an implementation of the standard library's `fill_n()` algorithm:

   ```
   template<OutputIterator<ValueType V> Iter, Integral Size, ValueType T>
   Iter fill_n(Iter first, Size n, const T& value) {
     while (n > 0) { *first++ = value; ++n; }
     return first;
   }
   ```

   One interesting aspect of this formulation is that makes the iterator type more important than the value type, because the iterator type is associated with the concept. It is unclear how well the syntax would apply when a concept has two equally-important type parameters, e.g., an Addable concept with both left and right types.

2. *Multiple constraints* on one type parameter cannot be directly expressed. Some additional syntax is needed, for instance, to express that a type `T` must model both Less Than Comparable and Copy Constructible. One possibility is to use the `&&` operator for composing requirements, as suggested in [SDR03b]. For example, one could express the requirements of an implementation of a `min()` function as follows:

   ```
   template<LessThanComparable && CopyConstructible T>
   T min(const T& x, const T& y) {
     return x < y? x : y;
   }
   ```

We propose a different syntax that does not view concepts as "types of types", but rather as constraints on types. All constraints on type parameters are placed in a `where` clause, in the following manner:

```
template<typeid T> where { LessThanComparable<T> }
bool equivalent(const T& x, const T& y) { /* ... */ }
```

This syntax retains the template header and adds a set of extra constraints via the `where` clause. Specifying multiple constraints and using multi-parameter concepts (such as the Output Iterator concept mentioned above) are intuitive in this syntax, as demonstrated by the two function templates below.

```
template<typeid T> where { LessThanComparable<T>, CopyConstructible<T> }
T min(const T& x, const T& y) { /* ... */ }

template<typeid Iter, typeid Size, typeid V>
  where { OutputIterator<Iter, V>, Integral<Size> }
Iter fill_n(Iter first, Size n, const V& value);
```

Additionally, non-concept-based constraints can be placed on template parameters in `where` clauses; see Sections 3.2.7 and 3.2.8.

[3] There are currently three kinds of template parameters: type, non-type, and template.

3.2.6 Associated types

In C++03, associated types are typically expressed using traits classes [Mye95], but checking of template definitions precludes the use of traits in type-safe templates because traits can be specialized in relatively unconstrained ways. First-class support for associated types would replace traits and permit checking of template definitions.

Associated types can be represented directly as types inside the concept. For example, the Forward Iterator concept's associated types are expressed as follows:

```
template<typeid Iter>
concept ForwardIterator {
  typename value_type;
  typename difference_type;
  typename reference;
  typename pointer;

  // more requirements...
};
```

Associated types can be accessed like member types of the concept, e.g., `ForwardIterator<X>::value_type` where `X` is a model of Forward Iterator. This usage is very similar to traits (in particular, `iterator_traits`) and reflects existing practice. References to associated types in generic functions do not require the `typename` keyword because the concept states that the member (e.g., `value_type`) must be a type[4].

An alternative to placing associated types in the concept definition is to make them type parameters. For instance, Stroustrup [Str03] defines the Forward Iterator concept with a type parameter `V` for the value type of the iterator. However, this does not express all of the associated types, which would require four template type parameters, e.g.,

```
template<typeid V, typeid Distance, typeid Ref, typeid Ptr>
concept ForwardIterator { /* ... */ };
```

Now, each generic function or class that uses Forward Iterator would have to declare parameters for all four of these associated types, even in the common cases where they are not used. These parameters must then be deducible or explicitly supplied by the user (e.g., through traits), because they are part of the signature of the generic function or class. The situation becomes worse with other concepts: Reversible Container [ISOI98, Table 67], for instance, requires nine associated types.

We have opted to express associated types as nested types within concepts, because it closely matches existing practice (traits) and does not require users to provide all associated types when using a concept.

3.2.7 Same-type constraints

A generic function that accepts several type parameters often requires that two types (often associated types) be equal. For instance, several standard library algorithms require the value types of different iterator types to be the same. To support such requirements, a `where` clause may contain *same-type constraints*, which assert that two types are always the same. Same-type constraints are written using the equality operator `==`. For example, consider the following concept-enabled standard library algorithm declaration:

```
template<typeid InputIterator1, typeid InputIterator2>
  where { InputIterator<InputIterator1>::value_type == InputIterator<InputIterator2>::value_type,
          LessThanComparable<InputIterator<InputIterator1>::value_type> }
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);
```

In this declaration, the types `InputIterator1` and `InputIterator2` are required to model the concept Input Iterator. The value types of these concepts must be equivalent, and model the Less Than Comparable concept.

3.2.8 Integral-constant constraints

Both concept and same-type constraints are essential to direct support for generic programming. However, some templates may require constraints not expressed using concepts or same-type constraints. For instance,

[4] `ForwardIterator<X>` is not a dependent type, because it refers to a concept.

we might define a minimum-value function for an array that requires the array to have at least one element:

```
template<typename T, std::size_t N> where { N > 0 }
const T& min_value(const std::array<T, N>& x);
```

Here, a `where` clause indicates the requirement `N > 0`. This requirement is easily checked by the compiler when the declaration is instantiated. The `enable_if` class template emulates this behavior in C++03; see Section B.5 for details. By permitting integral-constant constraints in `where` clauses we avoid the need for `enable_if` and provide a consistent method for specifying such constraints. Peter Dimov separately proposed a similar extension for restricted templates [Dim04]. Integral-constant constraints improve backward compatibility for generic libraries, because one can adapt existing traits-based code to our proposed concepts; Appendix A provides details.

Integral-constant constraints are checked when a template is used but do not aid in checking a template definition. This means that, for instance, the constraint `is_same<T, U>::value` is not equivalent to the same-type constraint `T == U`: both check that the two types are the same when the template is used, but only the latter allows the compiler to assume that the types are the same when type-checking the template definition.

4 Proposed language features

Two kinds of entities will be added to the C++ language: concepts and models. Also, templates are extended with a new kind of opaque type parameter, described in the next section, and an optional `where` clause, described in Section 4.5.

4.1 Opaque template type parameters

This proposal introduces a new kind of template parameter, the *opaque type parameter*. Technically, the difference between an opaque parameter and a normal type parameter is that an opaque parameter does not contribute to the set of template parameter dependent expressions in a template definition. In C++, name lookup, and hence type checking of dependent expressions, is delayed until instantiation. The addition of opaque parameters can decrease the number of dependent expressions and thereby increase the number of expressions that are type checked prior to instantiation. If a template only uses opaque parameters, there shall be no dependent expressions, and the entire definition is checked at the point of definition (not delayed until instantiation).

We suggest reusing the `typeid` keyword for purposes of distinguishing opaque type parameters from normal type parameters. The following is an example of a function template that uses an opaque type parameter.

```
template<typeid T>
T* ptr_id(T* x) { return x; }
```

Opaque parameters have the property that *any* type argument may be bound to it, and if the template definition type checks, then there shall be no instantiation-time compilation errors. Intuitively, this should be the case for `ptr_id()`, but there are some corner case types that invalidate this. For example, instantiating with a reference type, such as `T=int&`, would result in an error. The template author could constrain `T` using a `where` clause (see Section 4.5), and rule out reference types, but this is clumsy.

We resolve this issue by stating that function templates include an implicit constraint that the type arguments must result in a valid function declaration. The following declaration is invalid and therefore the template includes an implicit constraint on `T` that it cannot be a reference type.

```
int&* ptr_id(int&* x); // Ill-formed, so int& is not allowed for T
```

This rule has some interesting implications. Take the following definition of `vector` (Section 4.5 describes `where` clauses):

```
template<typeid T> where { CopyConstructible<T>, Assignable<T> }
class vector { ... };
```

A function template with `vector` appearing in the declaration will have the implicit constraint that the type argument for vector must model `CopyConstructible` and `Assignable`.

```
template<typeid U> // CopyConstructible<U> and Assignable<U> are implicit constraints
void foo(vector<U>& v) {
  U u(v[0]); // OK to use copy constructor here.
}
```

Another example of an implicit constraint implied by this rule is that pass-by-value parameters are assumed to model `CopyConstructible`. Thus, the following template is well formed.

```
template<typeid T> T identity(T x) { return x; }
```

4.2 Concepts

A concept is introduced by a concept definition, which consists of a template parameter list, the name of the concept, a refinement clause, and concept member specifications. Most concepts will have a single type parameter. The following defines the `InputIterator` concept, where `X` is a placeholder for a type that satisfies the requirements of `InputIterator`.

```
template<typeid X>
concept InputIterator { ... };
```

The scope of the template parameters includes the refinement clause and the concept member specifications. The refinement clause is introduced by a colon followed by a list of concept names with type arguments for each. The intuition is that a refinement adds the requirements of the refined concept to the refining concept.

```
template<typeid X>
concept BidirectionalIterator : ForwardIterator<X> { ... };
```

Concepts consist of four kinds of members: pseudo-signatures, associated types, nested requirements, and same-type constraints.

4.2.1 Pseudo-signatures

Pseudo-signatures express concept operations. During type checking, they serve two purposes. When checking a model definition, pseudo-signatures specify the operations required of the model. When checking a template definition, pseudo-signatures specify some of the operations that may be legally used in its body.

Syntactically, a pseudo-signature is a function declaration. A pseudo-signature may be followed by a function body, providing a default implementation to be used if a model does not define the function (see Section 4.3 for more details and an example). The following definition of the `EqualityComparable` concept includes two pseudo-signatures and provides a default implementation for the second.

```
template<typeid T>
concept EqualityComparable {
  bool operator==(const T&, const T&);
  bool operator!=(const T& x, const T& y) { return !(x == y); }
};
```

The requirement that a modeling type have a member function is expressed with a pseudo-signature qualified by the type. The following excerpt from the `Container` concept shows the pseudo-signature for the `empty()` member.

```
template<typeid X>
concept Container {
  ...
  bool X::empty() const;
  ...
};
```

A concept may require a constructor by using a type for the function name. The modeling type `T` need not be a class type: any type that may be constructed with the given signature is permitted. The `DefaultConstructible` concept includes a requirement for a constructor.

```
template<typeid T>
concept DefaultConstructible {
  T::T();
};
```

A concept may require a function template via a pseudo-signature template. The corresponding function in the model must have constraints that can be satisfied by the constraints of the pseudo-signature. The following is an example of a pseudo-signature template.

```
template<typename G>
concept MutableGraph : Graph<G> {
  ...
  template<typeid P> where { Predicate<P, edge> }
  void remove_edge_if(P p, G g);
  ...
};
```

Similarly, a concept may require a member function template by qualifying the pseudo-signature template with the model type, such as the constructor template for the `Sequence` concept.

```
template<typeid X>
concept Sequence : Container<X> {
  template<typeid Iter> where { InputIterator<Iter> }
  X::X(Iter first, Iter last);
  ...
};
```

4.2.2 Associated types

A concept may require that a model provide a type definition for a particular type name (e.g., with a `typedef`, a class declaration, or an `enum`). For example, the following `Graph` concept requires that a model declaration for `Graph` specify a type for `edge` and `vertex`.

```
template<typeid G>
concept Graph {
  typename edge;
  typename vertex;
  ...
};
```

The type name introduced by an associated type requirement may be used in the concept body and in refining concepts. Consider the following example.

```
template<typeid T>
concept A {
  typename s;
};
template<typeid U>
concept B : A<U> {
  void foo(s);
};
```

The use of type name `s` in concept `B` is valid because `B` refines `A`, which requires the associated type `s`.

A concept may provide a default for an associated type. If a model does not specify the type definition, then the model uses the default. The default type need not be well-formed if the model provides the type definition. The following `InputIterator` concept requires four associated types that default to nested type definitions in the iterator.

```
template<typeid Iter>
concept InputIterator
  : CopyConstructible<Iter>, Assignable<Iter>, EqualityComparable<Iter> {
  typename value_type = Iter::value_type;
  typename reference = Iter::reference;
  typename pointer = Iter::pointer;
  typename difference_type = Iter::difference_type;
  ...
};
```

4.2.3 Nested requirements

A concept may require certain types to model another concept with a `require` clause that names the concept and type arguments. For example, the `Container` concept requires that the `iterator` type satisfy the requirements of `InputIterator`. Any well-formed type expression, including template parameters and associated types, may be used as type arguments in a `require` clause.

```
template<typeid X>
concept Container {
  ...
  require typename X::iterator;
  require InputIterator<X::iterator>;
  ...
};
```

A nested requirement differs from a refinement in several respects:

- When a model of a concept is defined, there must already be model definitions (Section 4.3 describes model definitions) for the nested requirements, but not for the refinements. Consider the following example.

  ```
  template<typeid T>
  concept A { };

  template<typeid T>
  concept B : A<T> { };

  template<typeid T>
  concept C { require A<T>; };

  model C<int> { }; // Error, model A<int> must already be defined
  model B<int> { }; // OK, and model A<int> is implicitly generated
  model C<int> { }; // Now this is OK
  ```

  The first model definition `C<int>` is an error because there must already be a declaration for `A<int>`. The second model `B<int>`, however, is well-formed because `B` refines `A` and all of the requirement of `A` (there aren't any) are fulfilled by `B<int>`. The third model definition, for `C<int>`, is valid because the model `A<int>` is implicitly generated from model `B<int>` (see Section 4.3.2).

- Associated types of a concept being refined are associated types of the refining concept. The same is not true of nested requirements. Consider the following example.

  ```
  template<typeid T>
  concept A {
    typename s;
  };

  template<typeid T>
  concept B : A<T> {
    void foo(s); // OK, s is defined in concept A
  };

  template<typeid T>
  concept C {
    require A<T>;
    void foo(s); // Error
    void foo(A<T>::s); // OK, use model identifier to access associated type
  };
  ```

  The use of type name `s` in `B` is well-formed because `B` refines `A` and `s` is an associated type of `A`. The use of type name `s` in `C` is an error. However, the model identifier `A<T>` (see Section 4.3.3) can be used to access its associated type `s`.

- Refinements may affect function selection (see Section 4.5.3).

A concept may require a nested type by qualifying the type name with the parameter type. For example, the `Container` concept requires that a model type `X` have a nested `typedef` for `value_type`.

```
template<typeid X>
concept Container {
  require typename X::value_type;
  ...
};
```

4.2.4 Same-type constraints

A same-type constraint expresses the requirement that two type expressions denote the same type. Consider the following example.

```
template<typeid X>
concept Container {
  require typename X::value_type;
  require typename X::iterator;
  require InputIterator<X::iterator>;
  require InputIterator<X::iterator>::value_type == X::value_type;
  ...
};
```

The `value_type` of the container's iterator must be the same as the `value_type` of the container. The type expression

```
InputIterator<X::iterator>::value_type
```

refers to the `value_type` of the model of `InputIterator` for the type `X::iterator`.

4.3 Models

A model definition establishes that a type meets the requirements of a concept. A model definition consists of a concept identifier, template arguments, and optionally model member definitions. Consider the following example.

```
class student_record {
  string id;
  string name;
  string address;
  friend model EqualityComparable<student_record>;
};
model EqualityComparable<student_record> {
  bool operator==(const student_record& a, const student_record& b)
  { return a.id == b.id; }
};
```

A model of the `EqualityComparable` concept (from Section 4.2.1) is defined for `student_record`. The `EqualityComparable` concept has two requirements; this model satisfies the requirement for `operator==`, and then uses the default defined in `EqualityComparable` for `operator!=`, whose implementation invokes this `operator==`. A new kind of friend declaration is introduced, to allow a class to grant friendship to a model.

4.3.1 Checking a model with respect to its concept

All of the requirements of the modeled concept, and the concepts it refines (transitively according to the refinement relation) must be satisfied according to the following rules, otherwise a diagnostic error message is required. We first discuss associated types, then pseudo-signatures, nested requirements, and finally same-type constraints.

- A requirement for an associated type may be satisfied in one of the following ways.

    1. A type definition in the body of the model will satisfy an associated type requirement. For example, the following concept `A` requires an associated type `t`. The model provides a typedef with `bool` for `t`.

```
template<typeid T>
concept A {
  typename t;
};
model A<float> {
  typedef bool t;
};
```

2. If the associated type is from a refinement, and there is a model definition for the refinement, then the model need not (and may not) provide a typedef for that associated type. In the following example, concept `B` refines the concept `A` defined above. The model declaration for `B<float>` does not include a typedef for `t` since there is one in model `A<float>`.

```
template<typeid T>
concept B : A<T> { };

model B<float> { }
```

The following definition of model `B<float>` is ill-formed, since it tries to redefine `t`.

```
model B<float> {
  typedef char t; // Error, A<float> is already defined.
};
```

3. If the associated type is from a refinement, and there is not yet a model definition for the refinement, then the model must satisfy the requirement by providing a type definition. For example, below we define a model `B<double>` and assume there is no previous definition for model `A<double>`. The requirement for the associated type `t` from concept `A` is satisfied by the typedef in model `B<double>`.

```
model B<double> {
  typedef long t;
};
```

4. If there is a default for the associated type in the concept, then the model need not provide a type definition. In the example below, the associated type `t` for model `C<float>` will be `int`.

```
template<typeid T>
concept C {
  typename t = int;
};
model C<float> { };
```

- A pseudo-signature requirement in a concept may be satisfied by a model according to the following rules. A pseudo-signature may contain occurrences of the concept template parameters and associated types. To obtain the pseudo-signature that must be satisfied by the model the template arguments and associated types provided by the model are substituted for the concept's parameters and associated types. To illustrate, suppose the following definition of concept `A` and model `A<int>`.

```
template<typeid T>
concept A {
  typename s;
  s foo(T);
};
model A<int> {
  typedef char s;
  ...
};
```

The model `A<int>` must satisfy the requirement for a function with the signature

```
char foo(int);
```

This requirement may be satisfied according to the following rules.

1. A model may satisfy the pseudo-signature requirement with a function definition in the model body.

```
model A<int> {
  typedef char s;
  char foo(int x) { return 'a'; }
};
```

2. If the pseudo-signature is from a refinement, and there is a model definition for the refinement, then the model need not (and may not) provide a definition for the function.

```
template<typeid U>
concept B : A<T> { };

model B<int> { }; // no definition of foo() needed, because it is provided by model A<int>
```

3. If the pseudo-signature is not satisfied by one of the above rules (a definition in the model body or by a model of a refinement), then the pseudo-signature may be satisfied by a function in the enclosing scope of the model declaration. Function lookup is performed as for a function call expression whose function and arguments are given by the pseudo-signature. A forwarding function whose signature exactly matches the pseudo-signature is generated by the C++ implementation, and the body of this function consists of a function call to the result of the function lookup. The return type of the found function shall be convertible to the return type of the pseudo-signature, otherwise a diagnostic is required. In the following example, the requirement for `operator+` is satisfied by `X::operator+`.

```
template<typeid T>
concept C {
  bool operator+(const T&, const T&);
};
class X {
  bool operator+(const X&) const { return false; }
};
model C<X> { };
```

4. If the pseudo-signature is not satisfied by one of the above rules, then the default definition in the concept is used. Otherwise, a diagnostic is required. The example at the beginning of this section, of the `student_record` that models `EqualityComparable`, demonstrates the use of default implementations in concepts.

- A nested concept requirement must be satisfied by a previous model definition. Consider the following example.

```
template<typeid T>
concept A { };

template<typeid U>
concept B {
  require A<U>;
};

model A<int> { };
model B<int> { };
```

  Concept `B` contains the nested requirement for `A<U>`. The model `B<int>` is valid because there is a previous model definition for `A<int>`. Note that occurrences of template parameters and associated types in the nested requirement, such as `U` in `require A<U>;`, are replaced by the arguments and type definitions in the model. So `U` is replaced by `int`.

- A model satisfies a same-type constraint if the the two type expressions are equivalent types after concept template parameters and associated types have been replaced by the arguments and typedefs from the model. Consider the following example.

```
template<typeid T>
concept Bag {
  typename value_type;
  typename iterator;
  require typename iterator::value_type;
  require value_type == iterator::value_type;
};

struct int_iter { typedef int value_type; };
struct int_bag { };

model Bag<int_bag> {
  typedef int value_type;
  typedef int_iter iterator;
};
```

The `Bag` concept requires its associated `value_type` to be the same type as the nested `value_type` of its associated `iterator`. The model `Bag<int_bag>` satisfies this same type constraint because both of these types are equivalent to `int`.

4.3.2 Implicitly defined models

If there is a model definition for a concept that refines other concepts, and models for the refinements are not already defined, then model definitions for the refinements are implicitly generated. Consider the following example.

```
template<typeid T>
concept A {
  void foo();
};

template<typeid T>
concept B : A<T> { };

model B<int> {
  void foo() { }
};
```

Concept `B` refines `A`. There is a model definition for `B<int>`, but no explicit definition for `A<int>`. Thus, a model definition for `A<int>` is implicitly generated by the C++ implementation from the definition of `B<int>`.

4.3.3 Model identifiers

A concept name followed by template arguments is a model identifier. Model identifiers may be used to qualify access to entities in the scope of a model, such as type definitions and functions. In the example below, the model identifier `B<int>` is used to qualify `zero()`.

```
model B<int> {
  int zero() { return 0; }
};
int main() { return B<int>::zero(); }
```

4.3.4 Model templates

A model template establishes that a family of types models a concept. For example, the following model definitions establish pointers and pointers to constant values as models of `MutableRandomAccessIterator` and `RandomAccessIterator`.

```
template<typename T>
model MutableRandomAccessIterator<T*> {
  typedef T value_type;
  typedef T& reference;
  typedef T* pointer;
```

```
  typedef ptrdiff_t difference_type;
};

template<typename T>
model RandomAccessIterator<const T*> {
  typedef T value_type;
  typedef const T& reference;
  typedef const T* pointer;
  typedef ptrdiff_t difference_type;
};
```

In the upcoming Section 4.5 we extend templates with `where` clauses to express constraints on template parameters. The template may only be instantiated with arguments that satisfy the constraints. The following example demonstrates how constraints are useful in model templates.

```
template<typeid T, typeid Alloc> where { EqualityComparable<T> }
model EqualityComparable< vector<T, Alloc> > { };
```

A `vector` is a model of `EqualityComparable` if the value type `T` is `EqualityComparable`.

4.4 Concepts, models, and namespaces

A model must be defined in the same namespace as the concept being modeled. The following pattern will likely be the common case for concept and model definitions.

```
// Header file X.hpp
namespace X {
  template<typeid T> concept Drawable { void T::draw(); };
}

// Header file Y.hpp
#include "X.hpp"

namespace Y {
  struct shape { void draw(); };
}
namespace X {
  model Drawable<Y::shape>; // model declaration
}

// Source file Y.cpp
#include "Y.hpp"
void shape::draw() { ... }

namespace X {
  model Drawable<Y::shape> { }; // model definition
}
```

4.5 Template constraints

This proposal introduces constraints on templates in the form of a `where` clause. We give a brief introduction here to the syntax, and then discuss the meaning in the following sub-sections.

The syntax of template declarations (and definitions) is extended to include a `where` clause which consists of concept requirements, same-type constraints, and integral-constant expressions. A `where` clause follows the template header, starts with the `where` keyword, and is enclosed in braces. The constraints are comma-separated. The following is a simple example.

```
template<typeid T> where { Assignable<T>, CopyConstructible<T> }
void swap(T& a, T& b);
```

Template constraints play two roles in type checking:

1. When a template identifier is used, such as `vector<int>`, either the template's constraints shall be satisfied or a diagnostic is required.

2. When type checking the body of a template, the constraints add assumptions to the context of the compilation.

We discuss these roles in the next two sub-sections.

4.5.1 Concept checking

Consider the following example.

```
list<int> l;
sort(l.begin(), l.end()); // Error.
```

A diagnostic message shall be issued because `list<int>::iterator` is not a model of `RandomAccessIterator`.

For each concept constraint in the `where` clause, there must be a model declaration that best matches the type arguments, otherwise a diagnostic is required. The set of matching model declarations is determined in a similar fashion to how class specializations are chosen in (14.5.4.1) of the C++ standard. For each matching model template, the requirements in the model template's `where` clause must be satisfied, otherwise the model is removed from consideration. Rules similar to the partial ordering of class template specializations are then used to determine if there is a best match. Consider the following example.

```
template<typeid T>
concept A { };

template<typeid T> where { A<T> }
void foo(T) { }

template<typeid T>
model A<T*> { };

model A<int*> { };

int main() { int* x; foo(x); }
```

For the call to `foo()` there must be a model of `A<int*>`. Both model definitions match, but the second definition is a better match.

For same-type constraints, the implementation verifies that the two types are equivalent. Consider the following example.

```
template<typeid T, typeid U> where { T == U }
void foo(T t, U u) { }

int x;
foo(x, x); // OK, int == int
float z;
foo(x, z); // Error, int != float
```

When a template identifier appears inside a constrained template, there is a second phase of model resolution that determines which model will be used in the execution of the program. The second phase occurs when the enclosing template is instantiated and the bindings for all template parameters are known. The `where` clauses do not take part in this phase. Consider the following example.

```
template<typeid T>
concept Fooable {
  void foo(T x);
};

template<typeid T> where { Fooable<T> }
void bar(T t) { foo(t); }

model Fooable<int> {
  void foo(int x) { std::cout << "foobar!\n"; }
};

int main() { bar(2); } // The output is foobar!
```

The concept checking of the call to `foo()` is satisfied by the requirement `Fooable<T>` in the `where` clause of `bar()`. However, the final model resolution is performed once `bar()` is instantiated with `int` bound to `T`. Thus, the model `Fooable<int>` is chosen.

4.5.2 Type checking template definitions

In this section we discuss three aspects of type checking template definitions: name resolution, type equivalence, and the available model declarations.

Name resolution in a constrained template The `where` clause introduces names into the scope of the template definition. For each concept requirement, the pseudo-signatures from the concept and those concepts it refines or requires are in scope. Also, the associated types from the concept and the concepts it refines are in scope. The pseudo-signatures introduce function declarations, and the associated types introduce type names but not the actual type bindings (because they are not known until instantiation). For example, consider again the definition of `swap()`.

```
template<typeid T> where { Assignable<T>, CopyConstructible<T> }
void swap(T& a, T& b) {
  T tmp = a;
  a = b;
  b = tmp;
}
```

The concept requirement `Assignable<T>` brings the declaration

```
T& operator=(T&, const T&);
```

into scope for the body of `swap()`. This declaration allows the expressions `a = b` and `b = tmp` to type check.

The following example demonstrates how a `where` clause brings associated types into scope.

```
template<typeid C, typeid F>
  where { Container<C>, UnaryFunction<F, void, value_type> }
void for_each_c(const C& c, F f) {
  for (const_iterator i = c.begin(); i != c.end(); ++i)
    f(*i);
}
```

The type `value_type` and `const_iterator` are associated types from the `Container` concept.

If the same type name is introduced into scope from two or more concept constraints, then access to those types must be qualified by the model identifier. Consider the following example.

```
template<typeid C>
concept BackInsertionSequence : Sequence<C> { // and remember, Sequence refines Container
  reference C::back();
  const_reference C::back() const;
  void C::push_back(value_type);
  void C::pop_back();
};

template<typeid C1, typeid C2>
  where { Container<C1>, BackInsertionSequence<C2> }
void copy_c(const C1& c1, C2& c2) {
  for (Container<C1>::const_iterator i = c1.begin(); i != c1.end(); ++i)
    c2.push_back(*i);
}
```

The `const_iterator` associated type is introduced by both constraints, so the model identifier `Container<C1>` is used to disambiguate.

Type equivalence in a constrained template The same-type constraints of a `where` clause induce a partition of type expressions into equivalence classes. A same-type constraint `S == T` merges the equivalence classes that `S` and `T` belong to. Two types are considered equivalent if they are in the same equivalence class. We must specify rules about how type equivalence propagates through the structure of a type expression (the

type equality relation should be a congruence). For example, nested type expressions, such as `S::value_type` and `T::value_type`, are equivalent when the left-hand sides of the scope resolution operator are equivalent and the right-hand side identifiers are the same. Also, for any concept `A`, `A<S>::type` is equivalent to `A<T>::type` if `S` and `T` are equivalent. Further, if `B` is a refinement of `A`, then `B<U>::type` is equivalent to `A<S>::type` if `B<U>` refines (transitively) `A<T>` and `S` is equivalent to `T`.

Constraints add model declarations When type checking a template definition, each concept requirement in the `where` clause brings a model declaration into scope. Also, model declarations for the refinements and nested concept requirements are brought into scope (transitively according to refinements and nested requirements). Consider the following example.

```
template<typeid FwdIter1, typeid FwdIter2>
  where { ForwardIterator<FwdIter1>, ForwardIterator<FwdIter2> }
FwdIter1 search(FwdIter1 first1, FwdIter1 last1, FwdIter2 first2, FwdIter2 last2) {
  ...
  first1 = find(first1, last1, *first2);
  ...
}
```

The `find()` function requires its iterator argument to be a model of `InputIterator`, so this constraint must be satisfied in the above call to `find()`. The `search()` function includes `ForwardIterator<FwdIter1>` and `ForwardIterator<FwdIter2>` in its `where` clause. Thus model declarations for `ForwardIterator<FwdIter1>` and `ForwardIterator<FwdIter2>` are in scope for the body of `search()`. Also, because `ForwardIterator` is declared to refine `InputIterator`, model declarations for `InputIterator<FwdIter1>` and `InputIterator<FwdIter2>` are in scope. Thus, the call to `find()` in the body of `search()` is valid.

4.5.3 Concept-based function selection

The following are two possible approaches for concept-based function selection:

1. Extend the function overloading rules to take into account ordering of `where` clauses.
2. Introduce partial specialization of function templates, and specify how `where` clauses affect partial ordering of function template specializations.

More investigation will be needed to determine which of the above two approaches is the best. Nevertheless, we present some examples to show how we perceive `where` clauses should affect function selection. In the following example, the second `f` should be chosen because `B` refines `A`.

```
template<typeid T> concept A { };
template<typeid T> concept B : A<T> { };

template<typeid T> where { A<T> } void f(T x) { std::cout << "1"; }
template<typeid T> where { B<T> } void f(T x) { std::cout << "2"; }

model B<int>;
int main() { f(1); }
```

The output is:

```
2
```

In the next example, the second definition of `f` should be chosen because its constraints are a superset of the constraints of the first definition.

```
template<typeid T> concept A { };
template<typeid T> concept C { };

template<typeid T> where { A<T> } void f(T x) { std::cout << "1"; }
template<typeid T> where { A<T> , C<T> } void f(T x) { std::cout << "2"; }

model A<int>;
model C<int>;
int main() { f(1); }
```

The output is:

```
2
```

The following example should be ill-formed because the function call is ambiguous. The constraint `B<T>` in the first `f` is more specific than `A<T>` in the second `f`, but the second `f` has a second constraint that is not in the first `f`.

```
template<typeid T> concept A { };
template<typeid T> concept B : A<T> { };
template<typeid T> concept C { };

template<typeid T> where { B<T> } void f(T x) { std::cout << "1"; }
template<typeid T> where { A<T>, C<T> } void f(T x) { std::cout << "2"; }

model B<int>;
model C<int>;
int main() { f(1); } // ambiguous
```

4.5.4 Constrained class members

A class template member may be constrained with a `where` clause, even if the member is not a template. For example, the `sort()` member of the `list` class adds the further requirement on `T` that it model `LessThanComparable`.

```
template<typeid T, typeid Alloc = std::allocator<T> > where { CopyConstructible<T>, Allocator<Alloc, T> }
class list {
  ...
  where { LessThanComparable<T> } void sort();
  ...
};
```

5 Impact

This section describes the impact of the proposed changes on users, the standard library, and compiler vendors.

5.1 New keywords

This proposal introduces several new keywords into the C++ language. The new keywords are `concept`, `model`, `where`, and `require`. However, since each keyword poses a backward-compatibility problem, we can reuse existing keywords in new contexts:

- `where` could be replaced with `if`, `using`, or even a vertical bar (|) meaning "such that."
- `require` could be replaced be `where` or `if`.
- `model` could be replaced by a concept specialization syntax (similar to class template (partial) specialization). However, this is very likely to confuse users, because conceptually a model is not a concept but an implementation of a concept.
- `concept` could be replaced by some combination of keywords, such as `virtual class`. However, this syntax de-emphasizes the important differences between abstract classes and concepts, and may therefore lead to unnecessary confusion.

5.2 Impact on users

When a generic library written using C++03 is updated to use concepts, the way in which users interact with the library will change. The two major changes are as follows:

1. The compiler will provide improved diagnostics and type safety for templates, both uses and definitions.
2. The user will be required to introduce model declarations for their types.

The first item is a clear advantage: by introducing direct support for generic programming into C++03, the problems of weak type checking and horrible error messages can be eliminated, making generic libraries (and generic programming in general) more accessible. The second item is both an advantage and a disadvantage. It is advantageous because the introduction of explicit models adds additional type checking (i.e., checks whether the syntax of the model matches the syntax of the concepts) and asserts semantic properties that were otherwise only assumed to exist. The disadvantage is that porting will require some effort to write these model declarations. This disadvantage can be mitigated by careful generic library design (e.g., providing model templates that map from traits structures; see Appendix A), by compiler-supplied models for some concepts with simple semantics (e.g., Default Constructible for types with a public default constructor, Assignable for POD types that can be assigned), and by proper compiler diagnostics that describe what models are required.

5.3 Impact on the standard library

The C++ standard library is itself a generic library that could be extended to support concepts. This extension would require several specific changes, all of which are demonstrated in Section A:

1. *Replace the requirements tables and traits for concepts with new concept definitions.* This change formalizes the semantics of concepts and will likely eliminate ambiguities that have arisen due to the use of valid expressions in the descriptions and the lack of automated checking.
2. *Provide or require model definitions for all standard library components.* The standard already specifies when a particular library component meets the requirements of a concept; we need only state that these library components have model declarations for those concepts. In some cases, these will be model templates with constraints, e.g., `vector<T>` models Equality Comparable when `T` models Equality Comparable.
3. *Specify requirements via where clauses.* The templates in the standard library list requirements informally, e.g., by naming template parameters `RandomAccessIterator` when they must model Random Access Iterator. Replace these informal requirements with `where` clauses that convey the same information in a formal way. This will simplify the description of some parts of the standard, e.g., the "do the right thing" clause for the container constructors taking iterators. In addition, the improved type checking will verify the specification and most likely unearth bugs that have been lurking in the standard.
4. *Provide model definitions for backward-compatibility.* The conventions of the standard library, such as `iterator_traits`, can be used to build model templates that will greatly simplify porting from the existing standard library to a concept-enabled one. These models will be required in order to provide maximal backward compatibility, i.e., to minimize the number of changes users will need to make to recompile their code with the new standard library. For instance, the following model template adapts all existing random access iterators to the `RandomAccessIterator` concept:

```
template<typename OldIter>
where {Convertible<typename std::iterator_traits<OldIter>::category*,
                   std::random_access_iterator_tag*>}
model RandomAccessIterator<OldIter> {};
```

The most interesting question regarding changes to the standard library revolves around what we can do with components that have inconsistencies in their specification. For instance, `vector<bool>` does not model the same concepts as the primary template for `vector` because its iterators do not model the Random Access Iterator concept: an error that would have been caught given the extensions in this proposal. We expect that a full review of the changes to be made to the standard library will uncover additional, subtle problems with the specification of the library.

5.4 Impact on compiler vendors

The extensions proposed here are numerous and will undoubtedly require a nontrivial amount of effort to implement in any compiler. However, we have taken great care to ensure that this proposal retains the existing template compilation model, extending it without requiring fundamental changes. We describe the expected impact of the crucial features of this proposal on a compiler:

- *Concept syntax*: Concept syntax will require changes to the parser, because although concept definitions are similar to class definitions, they are not the same. Granted, much of the parsing logic for class templates can be reused here. Since there is very little extra checking required for concept definitions, this is not expected to be complicated. Otherwise, the representation of concepts will be similar to that of class templates.
- *Model syntax*: Model syntax will require some effort to parse. Models are more complex to parse than concepts because their definitions are checked at definition time. The representation of models, however, may be similar to that of a specialization of a class template by using the same lookup and ordering rules.
- *Where clauses*: `where` clauses require changes to two distinct parts of the compiler. The first is in template definitions, where the compiler must introduce the syntax of concepts appearing in the `where` clause into the lexical scope of the template. The second change is at the instantiation point of a template constrained by a `where` clause, which involves searching for and instantiating appropriate models. This latter step is similar to selecting the best class template (partial) specialization, and should be relatively simple to implement.
- *Parsing templates*: Template parsing changes primarily because `where` clauses introduce extra operations and types into the scope, and because more type checking is performed when a template is initially defined. Here, the change occurs when dependent contexts become non-dependent contexts, enabling more checking in the earlier phase.
- *Function selection and partial ordering*: The function overloading and/or partial ordering rules must be augmented to include rules for concept refinement comparisons. These should be very similar to rules for derived-to-base pointer conversion comparisons.

6 Acknowledgements

The formulation of concepts for C++ presented here was greatly influenced by discussions with David Abrahams. We are grateful to Bjarne Stroustrup, Gabriel Dos Reis, and Mat Marcus for their valuable input. We thank Matthew Austern for his work on the SGI STL documentation which inspired many aspects of this proposal. We thank Alexander Stepanov and David Musser for bringing generic programming into the C++ community. This work was supported by a grant from the Lilly Endowment. The fourth author was supported by a Department of Energy High Performance Computer Science Fellowship.

A Example: Standard library concepts and declarations

To demonstrate that this proposal is sufficient to express the type requirements of the C++ standard library, we present some examples from the standard library, rewritten to use concepts. The examples were chosen

to stress-test the proposal; diversity of features was the goal in selection. Wherever the standard provides a requirements table to specify a concept, the requirements table is provided side-by-side with the definition of the concept. Table numbers refer to the table numbers in the standard.

A.1 Helper concepts

The definition of the Less Than Comparable concept demonstrates the use of default implementations for concept operations. Here, default implementations serve the same purpose as the comparison operators in the `std::rel_ops` namespace (but without its problems).

Table 29, Less Than Comparable requirements [ISOI98]

expression	return type
`a < b`	convertible to `bool`

Type `T` is a model of Less Than Comparable and `a`, `b` are values of type `T`.

```
template<typeid T>
concept LessThanComparable {
  bool operator<(T a, T b);
  bool operator>(T a, T b) {return b < a;}
  bool operator<=(T a, T b) {return !(b < a);}
  bool operator>=(T a, T b) {return !(a < b);}
};
```

The definition of Copy Constructible illustrates pseudo-signatures for constructors and destructors. We also see how it can sometimes require fewer pseudo-signatures than valid expressions to express a concept.

Table 30, Copy Constructible requirements [ISOI98]

expression	return type
`T(t)`	
`T(u)`	
`T::~T()`	
`&t`	`T*`
`&u`	`const T*`

Type `T` is a model of Copy Constructible, `t` is a value of type `T` and `u` is a value of type `const T`.

```
template<typeid T>
concept CopyConstructible {
  T::T(const T&);
  T::~T();
  T* operator&(T&);
  const T* operator&(const T&);
};
```

A.2 Iterator concepts

Concepts in the iterator hierarchy demonstrate several features of the proposal, such as associated types and concept refinement. Before we can express the iterator concepts, a helper concept Arrowable is needed to express the iterative behavior of the `->` operator. The `Ptrlike` type parameter refers to an object that can be on the left-hand side of the `->` operator, e.g., a pointer or an object with an overloaded `operator->`. The `Value` type parameter is the return type produced by following the chain of `->` operators.

```
template<typeid Ptrlike, typeid Value>
concept Arrowable {
  typename arrow_result;
  arrow_result operator->(Ptrlike);
  require Arrowable<arrow_result, Value>;
};
```

This concept is particularly interesting because it is recursive: the result of `operator->` must itself model Arrowable. This behavior mimics the C++ arrow operator, which permits proxies so long as the final result is a pointer (i.e., the base case). The following model templates for pointers provide the base case for Arrowable:

```
template<typeid T>
model Arrowable<T*, T&> {
  typedef T* arrow_result;
  T* operator->(T* x) { return x; }
};
```

```
template<typeid T>
model Arrowable<const T*, const T&> {
  typedef const T* arrow_result;
  const T* operator->(const T* x) { return x; }
};
```

Equipped with the Arrowable concept, we can now write the iterator concepts. We choose to define a helper concept Iterator Associated Types for the associated type requiremens common to both Input Iterator and Output Iterator. The Iterator Associated Types concept demonstrates how associated types can have default definitions:

```
template<typeid Iter>
concept IteratorAssociatedTypes {
  typename value_type = Iter::value_type;
  typename difference_type = Iter::difference_type;
  typename pointer = Iter::pointer;
  typename reference = Iter::reference;
};
```

Because the standard definitions of the Input Iterator concept and its refinements require the result of the postfix ++ operator to be dereferenceable but not incrementable, a special concept is defined for this purpose. It is assumed that there is a Convertible concept defined.

```
template<typeid Iter>
concept ReadableIterator: IteratorAssociatedTypes<Iter> {
  reference operator*(Iter);
  require Convertible<reference, value_type>;
};
```

In Input Iterator, the postincrement operator can return a proxy. Because of the pseudo-signatures used in this proposal, a new associated type must be defined to represent the proxy type. Otherwise, this example is straightforward, but please note the use of the Arrowable concept.

Table 73, Input Iterator requirements [ISOI98]

operation	type
`X u(a);`	`X`
`u = a;`	`X&`
`a == b`	convertible to `bool`
`a != b`	convertible to `bool`
`*a`	convertible to `T`
`a->m`	
`++r`	`X&`
`(void)r++`	
`*r++`	convertible to `T`

Type `X` is a model of Input Iterator, `u`, `a`, and `b` are values of type `X`, type `T` is a value type of iterator `X`, `m` is the name of a member of type `T`, and `r` is a reference to a non-constant `X` object.

```
template<typeid Iter>
concept InputIterator: EqualityComparable<Iter>,
                       CopyConstructible<Iter>,
                       Assignable<Iter>,
                       ReadableIterator<Iter> {
  pointer operator->(Iter);
  require Arrowable<pointer, value_type>;
  Iter& operator++(Iter&);
  typename postincrement_result; // Internal proxy type
  postincrement_result operator++(Iter&, int);
  require ReadableIterator<postincrement_result>;
};
```

The Output Iterator concept has a second type parameter to represent the value type, since a single output iterator type can accept many different value types. In particular, given an output iterator `Iter` that is able to directly store objects of type `T` (i.e., its dereference operation returns a type which has an assignment operator taking a parameter of type `T`), objects of any type `U` where `U` is convertible to `T` are also valid for storage by the iterator. In order to handle the common case of an iterator directly taking only one type, a special concept Basic Output Iterator is defined, along with a model template providing the correct conversion behavior. This model template states that for every model `Iter` of Basic Output Iterator and any type `Value` which is convertible to the type storable in `Iter`, `Iter` and `Value` together model Output Iterator. This one model template expresses this relationship for every possible iterator and value type.

Table 74, Output Iterator requirements [ISOI98]

operation	type
`X(a)`	
`X u(a);` `u = a;`	
`*r = o`	result is not used
`++r`	`X&`
`r++`	convertible to `const X&`
`*r++ = o`	result is not used

Type `X` is a model of Output Iterator, `u` and `a` are values of type `X`, `o` is a value whose type is in the value type set for type `X`, and `r` is a reference to a non-constant `X` object.

```
template<typeid Iter, typeid Value>
concept OutputIterator: IteratorAssociatedTypes<Iter> {
  reference operator*(Iter& i);
  void operator=(reference ref, Value v);
};

template<typeid Iter>
concept BasicOutputIterator
  : IteratorAssociatedTypes<Iter> {
  reference operator*(Iter& i);
  void operator=(reference ref, value_type v);
};

template<typeid Iter, typeid Value>
where {
  BasicOutputIterator<Iter>,
  Convertible<Value,
              BasicOutputIterator<Iter>::value_type> }
model OutputIterator<Iter, Value> {
  typename reference =
             BasicOutputIterator<Iter>::reference;
  reference operator*(Iter& i) {
    return *i; // Uses version from BasicOutputIterator
  }
  void operator=(reference ref, Value v) {
    ref = (BasicOutputIterator<Iter>::value_type)v;
  }
};
```

For Forward Iterator and all iterator concepts refining it, there are two variants of each concept: mutable and non-mutable forms. The reference type requirement expressed in the standard (that the reference type of a forward iterator be either `const value_type&` or `value_type&`) is not expressible using our proposal, since we do not propose disjunctive constraints. Much of the complication in these concepts is because a single table of iterator requirements in the standard really defines both the mutable iterator concept and the non-mutable iterator concept.

Table 75, Forward Iterator requirements [ISOI98]

operation	type
`X u;`	
`X()`	
`X(a)`	
`X u(a);` `X u = a;`	
`a == b`	convertible to `bool`
`a != b`	convertible to `bool`
`r = a`	`X&`
`*a`	`T&` if `X&` is mutable, otherwise `const T&`
`a->m`	`U&` if `X` is mutable, otherwise `const U&`
`r->m`	`U&`
`++r`	`X&`
`r++`	convertible to `const X&`
`*r++`	`T&` if `X` is mutable, otherwise `const T&`

Type `X` is a model of Forward Iterator, `u`, `a`, and `b` are values of type `X`, type `T` is a value type of iterator `X`, `m` (with type `U`) is the name of a member of type `T`, and `r` is a reference to a non-constant `X` object.

```
template<typeid Iter>
concept ForwardIterator: InputIterator<Iter>,
                         DefaultConstructible<Iter> {
  require Convertible<reference, const value_type&>;
  require Arrowable<pointer, const value_type&>;
  require postincrement_result == const Iter&;
};

template<typeid Iter>
concept MutableForwardIterator
  : ForwardIterator<Iter>,
    BasicOutputIterator<Iter> {
  require reference == value_type&;
  require Arrowable<pointer, value_type&>;
};
```

The remaining iterator concepts require relatively straightforward translations, requiring no new features. Again, the standard's requirements tables expression both mutable and non-mutable versions of the concepts, which we present as separate concepts.

Table 76, Bidirectional Iterator requirements [ISOI98]

operation	type
`--r`	`X&`
`r--`	convertible to `const X&`
`*r--`	convertible to `T`

Type `X` is a model of Bidirectional Iterator, `T` is the value type of `X`, and `r` is a non-constant reference to an `X`.

```
template<typeid Iter>
concept BidirectionalIterator: ForwardIterator<Iter> {
  Iter& operator--(Iter&);

  Iter& operator--(Iter& i, int)
  { Iter temp = i; --i; return temp; }
};

template<typeid Iter>
concept MutableBidirectionalIterator
  : MutableForwardIterator<Iter>,
    BidirectionalIterator<Iter> {
  require reference == value_type&;
  require Arrowable<pointer, value_type&>;
};
```

Table 77, Random Access Iterator requirements [ISOI98]

operation	type
`r += n`	`X&`
`a + n` `n + a`	`X`
`r -= n`	`X&`
`a - n`	`X`
`b - a`	`Distance`
`a[n]`	convertible to `const T&`
`a < b`	convertible to `bool`
`a > b`	convertible to `bool`
`a >= b`	convertible to `bool`
`a <= b`	convertible to `bool`

Type `X` is a model of Random Access Iterator, `T` is the value type of `X`, `a` and `b` are values of type `X`, coder is a non-constant reference to an `X`, `Distance` is the difference type of `X`, and `n` is a value of type `Distance`.

```
template<typeid Iter>
concept RandomAccessIterator
  : BidirectionalIterator<Iter>,
    LessThanComparable<Iter> {
  Iter& operator+=(Iter& i, difference_type);

  Iter& operator-=(Iter& i, difference_type d)
  {return (iter += (-d));}

  Iter operator+(Iter i, difference_type d)
  {i += d; return i;}

  Iter operator+(difference_type d, Iter i)
  {i += d; return i;}

  Iter operator-(Iter i, difference_type d)
  {i -= d; return i;}

  difference_type operator-(Iter i, Iter j);

  const value_type&
  operator[](Iter i, difference_type d)
  {return *(i + d);}
};

template<typeid Iter>
concept MutableRandomAccessIterator
  : MutableBidirectionalIterator<Iter>,
    RandomAccessIterator<Iter> {
  require reference == value_type&;
  require Arrowable<pointer, value_type&>;
};
```

To improve backward compatibility, a set of model templates should be provided by the standard library to adapt existing iterators (based on `iterator_traits`) to the new iterator concepts. Each model template is constrained by a `where` clause that checks the iterator category against the standard tag clauses to determine which standard library iterator concepts it models. For `Forward Iterator` and the concepts that refine it, mutability is determined by comparing the iterator's reference type against `value_type&`.

```
template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::input_iterator_tag*> }
model InputIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::output_iterator_tag*> }
model OutputIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::forward_iterator_tag*> }
model ForwardIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::forward_iterator_tag*>,
        is_same<typename std::iterator_traits<OldIter>::reference,
                typename std::iterator_traits<OldIter>::value_type&>::value }
model MutableForwardIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::bidirectional_iterator_tag*> }
```

```
model BidirectionalIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::bidirectional_iterator_tag*>,
        is_same<typename std::iterator_traits<OldIter>::reference,
                typename std::iterator_traits<OldIter>::value_type&>::value }
model MutableBidirectionalIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::random_access_iterator_tag*> }
model RandomAccessIterator<OldIter> {};

template<typename OldIter>
where { Convertible<typename std::iterator_traits<OldIter>::category*,
                    std::random_access_iterator_tag*>,
        is_same<typename std::iterator_traits<OldIter>::reference,
                typename std::iterator_traits<OldIter>::value_type&>::value }
model MutableRandomAccessIterator<OldIter> {};
```

A.3 Container concepts

The Sequence concept shows that a concept can require polymorphic functions, including member functions and constructors.

Table 68, Sequence requirements [ISOI98]

expression	return type
`X(n, t)` `X a(n, t)`	
`X(i, j)` `X a(i, j)`	
`a.insert(p, t);`	`iterator`
`a.insert(p, n, t);`	`void`
`a.insert(p, i, j);`	`void`
`a.erase(q);`	`iterator`
`a.erase(q1, q2);`	`iterator`
`a.clear();`	`void`
`a.assign(i, j);`	`void`
`a.assign(n, t);`	`void`

Type `X` is a model of Sequence, `a` is a value of type `X`, `n` is a value of type `X::size_type`, `t` is a value of type `X::value_type`, `p` is a valid iterator of `a`, `q` is a dereferenceable iterator of `a`, `[q1, q2)` is a valid range in `a`, `i` and `j` denote iterators satisfying the input iterator requirements, and `[i, j)` denotes a valid range.

```
template<typeid X>
concept Sequence: Container<X> {
  X::X(size_type n, value_type t);

  template<typeid InputIter>
  where { InputIterator<InputIter>,
          Convertible<InputIterator<InputIter>::value_type,
                      value_type> }
  X::X(InputIter a, InputIter b);

  iterator X::insert(iterator p, value_type t);
  void X::insert(iterator p, size_type n, value_type t);

  template<typeid InputIter>
  where { InputIterator<InputIter>,
          Convertible<InputIterator<InputIter>::value_type,
                      value_type> }
  void X::insert(iterator p, InputIter a, InputIter b);

  iterator X::erase(iterator q);
  iterator X::erase(iterator q1, iterator q2);
  void X::clear();

  template<typeid InputIter>
  where { InputIterator<InputIter>,
          Convertible<InputIterator<InputIter>::value_type,
                      value_type> }
  void X::assign(InputIter i, InputIter j);

  void X::assign(size_type n, value_type t);

  require MutableForwardIterator<iterator>;
  require ForwardIterator<const_iterator>;
};
```

A.4 Models

The `stack` definition demonstrates that class templates can also be concept-constrained, including using same-type constraints to express that the container used in a stack must be able to store the same type as the stack stores.

```
template<typeid T, typeid Seq = std::deque<T> >
where { CopyConstructible<T>,
        Assignable<T>,
        BackInsertionSequence<Seq>, // Represents requirements in lib.stack
        typename Seq::value_type == T }
class stack {
  public:
  typedef typename Seq::value_type value_type;
  typedef typename Seq::reference reference;
  typedef typename Seq::const_reference const_reference;
  typedef typename Seq::size_type size_type;
  typedef Seq container_type;

  protected:
  Seq c;

  public:
  explicit stack(const Seq& = Seq());
  stack(const stack&);
  stack& operator=(const stack&);
  bool empty() const;
  size_type size() const;
  value_type& top();
  const value_type& top() const;
  void push(const value_type&);
  void pop();
};

template<typeid T, typeid Seq>
model CopyConstructible<stack<T, Seq> > {};

template<typeid T, typeid Seq>
model Assignable<stack<T, Seq> > {};

template<typeid T, typeid Seq>
model DefaultConstructible<stack<T, Seq> > {};
```

A `stack` will have a well-defined equality comparison operator (`==`) when its element type models Equality Comparable. In this case, the `stack` itself will also model Equality Comparable. The same logic applies to the less-than operator (`<`) and the Less Than Comparable concept. These properties of `stack` can be expressed with the following model templates and generic functions:

```
template<typeid T, typeid Seq>
where { EqualityComparable<T> }
bool operator==(const stack<T, Seq>&, const stack<T, Seq>&) { ... }

template<typeid T, typeid Seq>
where { EqualityComparable<T> }
model EqualityComparable<stack<T, Seq> > { ... };

template<typeid T, typeid Seq>
bool operator<(const stack<T, Seq>&, const stack<T, Seq>&)
where { LessThanComparable<T> } { ... }

template<typeid T, typeid Seq>
where { LessThanComparable<T> }
model LessThanComparable<stack<T, Seq> > {};
```

A.5 Algorithms

This section contains declarations of several standard library algorithms, to illustrate how the introduction of `where` clauses into the standard library would affect the presentation. The definition of one of the most basic algorithms, `copy()`, follows:

```
template<typeid InputIter, typeid OutputIter>
where { InputIterator<InputIter>,
        OutputIterator<OutputIter, InputIterator<InputIter>::value_type> }
OutputIter copy(InputIter a, InputIter b, OutputIter out) {
  while (a != b) *out++ = *a++;
  return out;
}
```

The unary `transform()` algorithm introduces function objects, which are identified by the Callable concept family. The numbered concepts Callable0, Callable1, Callable2, etc., require that the first type parameter be an object that can be called with a given set of parameter types (the rest of the type parameters to the concept). Function pointer types and classes with overloaded `operator()`s are examples of Callable types.

```
template<typeid InputIter, typeid OutputIter, typeid Func>
where { InputIterator<InputIter>,
        Callable1<Func, value_type>,
        OutputIterator<OutputIter, result_type> }
OutputIter transform(InputIter a, InputIter b, OutputIter out, Func f) {
  while (a != b) *out++ = f(*a++);
  return out;
}
```

The binary `transform()` algorithm is the first algorithm to have multiple input iterator types as parameters. Since each iterator type has a `value_type`, the algorithm qualifies references to `value_type`. The Callable2 concept is used to refer to a binary function object.

```
template<typeid InputIter1, typeid InputIter2, typeid OutputIter, typeid Func>
where { InputIterator<InputIter1>,
        InputIterator<InputIter2>,
        Callable2<Func, InputIterator<InputIter1>::value_type,
                        InputIterator<InputIter2>::value_type>,
        OutputIterator<OutputIter, result_type> }
OutputIter transform(InputIter1 a, InputIter1 b,
                     InputIter2 c, OutputIter out, Func f) {
  while (a != b) *out++ = f(*a++, *c++);
  return out;
}
```

Same-type constraints are required by several standard library algorithms, especially those that involve comparing two sequences. The following declaration of the `includes()` algorithm requires that the two input iterator sequences have the same `value_type`.

```
template<typeid InputIter1, typeid InputIter2,
         typeid Cmp = std::less<InputIterator<InputIter1>::value_type> >
where { InputIterator<InputIter1>,
        InputIterator<InputIter2>,
        InputIterator<InputIter1>::value_type == InputIterator<InputIter2>::value_type
        StrictWeakOrdering<Cmp, InputIterator<InputIter1>::value_type> }
bool includes(InputIter1 a, InputIter1 b, InputIter2 c, InputIter2 d,
              Cmp cmp = Cmp());
```

The `advance` function demonstrates the use of concept-based function selection, by providing multiple definitions with different `where` clauses.

```
template<typeid Iter, typeid Distance>
where { InputIterator<Iter>,
        IntegralType<Distance>,
        Convertible<Distance, difference_type> }
void advance(Iter& i, Distance n) {
  while (n != 0) {++i; --n;}
}
```

```
template<typeid Iter, typeid Distance>
where { BidirectionalIterator<Iter>,
        IntegralType<Distance>,
        Convertible<Distance, difference_type> }
void advance(Iter& i, Distance n) {
  while (n > 0) {++i; --n;}
  while (n < 0) {--i; ++n;}
}

template<typeid Iter, typeid Distance>
where { RandomAccessIterator<Iter>,
        IntegralType<Distance>,
        Convertible<Distance, difference_type> }
void advance(Iter& i, Distance n) {
  i += n;
}
```

B C++ tricks obsoleted by this proposal

This proposal obsoletes many template tricks that are often used to simulate generic programming in C++, both in standard library implementations and libraries such as Boost. We do not expect these techniques to disappear entirely, because they may still be needed for template metaprogramming, but they will no longer be required for generic programming itself. Descriptions of several of these tricks—and the proposed extensions that will eliminate much of the need for them—follow.

B.1 Type traits

Type traits describe certain attributes of types, which often involve associated types and a "tag" class that states what concepts the types model. The prototypical example is `iterator_traits`, from the standard library, which is written as follows:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag : input_iterator_tag, output_iterator_tag {};
struct bidirectional_iterator_tag : forward_iterator_tag {};
struct random_access_iterator_tag : bidirectional_iterator_tag{};

template<typename T>
struct iterator_traits {
  typedef typename T::iterator_category iterator_category;
  typedef typename T::value_type        value_type;
  typedef typename T::difference_type   difference_type;
  typedef typename T::reference         reference;
  typedef typename T::pointer           pointer;
};
```

The value type of an iterator `Iter` is accessed as `typename iterator_traits<Iter>::value_type`. To determine what concepts the iterator models, the `iterator_category` type can be queried and it can be checked for convertibility to any of the `_tag` classes. However, the use of `iterator_traits<Iter>` implies that the type `Iter` is actually an iterator, otherwise the instantiation of `iterator_traits<Iter>` will likely fail.

The introduction of concepts and models in this proposal obsoletes the need for traits. The refinement hierarchy is encoded explicitly, as are associated types. Failures will only occur when the user has provided a model that is incomplete or incorrect.

B.2 Tag dispatching

Tag dispatching is a mechanism for selecting different implementations of a generic function based on what concepts the type arguments model. It is a longstanding trick for simulating concept-based function selection,

used for example to provide a constant time implementation of `advance()` for Random Access Iterators and a linear time implementation for Bidirectional Iterators. The `advance()` function can be implemented as follows:

```
template<typename Iter>
void advance_impl(Iter& x, typename iterator_traits<Iter>::difference_type n,
                  bidirectional_iterator_tag) {
  typedef typename iterator_traits<Iter>::difference_type difference_type;
  while (n > difference_type(0)) { ++x; --n; }
  while (n < difference_type(0)) { --x; ++n; }
}

template<typename Iter>
void advance_impl(Iter& x, typename iterator_traits<Iter>::difference_type n,
                  random_access_iterator_tag) {
  x += n;
}

template<typename Iter>
void advance(Iter& x, typename iterator_traits<Iter>::difference_type n) {
  typedef typename iterator_traits<Iter>::iterator_category category;
  advance_impl(x, n, category());
}
```

The `iterator_category` from the `iterator_traits` data structure determines which concept(s) the iterator conforms to, and then overload resolution picks the most refined concept (represented by the most derived class).

The extensions in this proposal obsolete tag dispatching because they directly support concept-based function selection. Thus, two versions of `advance()` would be provided, one accepting Random Access Iterators and the other accepting Bidirectional Iterators.

B.3 Concept checking

Concept checking [Sie00, SL00] involves moving the syntactic checking of template type parameters earlier in templates, so that incorrect *uses* of templates are detected earlier and with a shorter call stack. Concept checking relies on the library developer to create concept-checking classes that exercise all valid expressions required by concepts, e.g.,

```
template<typename X>
struct EqualityComparableChecker {
  void constraints(X x, X y) {
    bool b1 = (x == y);
    bool b2 = (x != y);
  }
};
```

By forcing instantiation of the member function `EqualityComparableChecker<T>::constraints`, one can check that a type `T` satisfies the syntactic requirements of the Equality Comparable concept. A generic function requiring its type parameter to be Equality Comparable would have such an instantiation at the top of the function body.

Concept checking is built into the notion of modeling a concept in this proposal; when a model declaration states that a type, or set of types, model a concept, the compiler verifies that the types satisfy the syntactic requirements of that concept. Furthermore, concept checks are part of function and class bodies, whereas `where` clauses are part of their interfaces. Thus, concept-based function selection and concept-based partial ordering become possible.

B.4 Concept archetypes

Concept archetypes [Sie00, SL00] aid in verifying that template *definitions* are correct with respect to concepts. A concept archetype is a type that fulfills only the minimal requirements of that concept, and as such any attempt to use operations not defined by the concept will fail when the template is instantiated with the archetype. For instance, the following is an archetype for the Equality Comparable concept:

```
class EqualityComparableArchetype {
  // Not CopyConstructible, Assignable, DefaultConstructible
  EqualityComparableArchetype();
  EqualityComparableArchetype(const EqualityComparableArchetype&);
  EqualityComparableArchetype& operator=(const EqualityComparableArchetype&);

public:
  friend bool
  operator==(const EqualityComparableArchetype&, const EqualityComparableArchetype&)
  { return true; }

  friend bool
  operator!=(const EqualityComparableArchetype&, const EqualityComparableArchetype&)
  { return true; }
};
```

Attempting to instantiate an algorithm that requires Equality Comparable but also requires a copy constructor with the type `EqualityComparableArchetype` will fail, because this type only meets the requirements of Equality Comparable. Developing proper archetypes is not trivial: for instance, the `bool` return types should be replaced with private types (different ones for the two functions) that are convertible to `bool` and the parameters to `operator==` could be replaced with yet more private types that can be implicitly constructed from expressions of type `const EqualityComparableArchetype&`.

This proposal obsoletes concept archetypes because it offers improved type checking for template definitions. The `where` clause of a template introduces the minimal syntax required, and no other syntax can be used within the function template. Thus, instead of requiring one to carefully craft archetype classes and provide separate instantiation tests that use these archetypes, the same checking will be performed by the compiler with no additional work on the part of the user.

B.5 `enable_if` and Substitution Failure Is Not An Error

Substitution Failure Is Not An Error (SFINAE) refers to a feature of C++ templates that permits templates to be removed from consideration because considering them would produce malformed types. This feature is used in the implementation of several type traits (e.g., `is_class` and `has_member`), but is most prominent in the `enable_if` class template, defined as follows:

```
template<bool Cond, typename T = void> struct enable_if { typedef T type; };
template<typename T> struct enable_if<false, T> { };
```

`enable_if` can be used to selectively enable or disable templates. For instance, the standard containers have constructors that take an iterator range, but these constructors can be accidentally selected over the repeated value constructor when the container stores an integral type. The C++03 definitions of these constructors are:

```
// Range constructor
template<typename InputIterator>
  vector(InputIterator first, InputIterator last);

// Repeated value constructor
vector(size_type n, const value_type& x = value_type());
```

The "do the right thing" clause of the standard requires that the range constructor check if the type `InputIterator` is integral and, if so, emulate the behavior of the repeated value constructor. This can be succinctly encoded via `enable_if` in the range constructor as:

```
// Improved range constructor
template<typename InputIterator>
  vector(InputIterator first, InputIterator last,
         typename enable_if<(!is_integral<InputIterator>::value), void*>::type = 0);
```

The `is_integral` trait will be true when the parameter is an integral type, so the first parameter to `enable_if` will be true only when `InputIterator` is *not* an integral type. In this case, we pick the primary `enable_if` template and the range constructor will be selected. However, if `InputIterator` is an integral type, we pick the partial specialization of `enable_if`, which does not contain a nested type `type`. Due to SFINAE,

the lack of nested `type` causes type deduction to fail for this overload, so it will not be considered and we automatically fall back to the repeated value constructor. Prior implementations of the "do the right thing" clause typically employed tag dispatching.

The `enable_if` construct is obsoleted by the `where` clause, which allows the use of arbitrary compile-time expressions to restrict the use of templates. The `where` clause also allows one to express constraints more directly; for instance, the range constructor should really have stated that it requires input iterators, but could not because attempts to instantiate it with types that are not iterators would cause an error (instead of silently selecting a different overload).

B.6 Barton & Nackman trick

The Barton & Nackman trick [BN94] is a way to provide several distinct implementations of the same concept but retain a common type signature. Using the Barton & Nackman trick, one can define a class template that is parameterized by the name of its derived class[5]. Its operations then cast to the derived class and forward the operation to that class. For instance, consider the implementation of a simple `Matrix` class:

```
// Interface template
template<typename T, typename Derived>
class Matrix {
public:
  Matrix(const Matrix& other)
  { static_cast<Derived*>(this)->assign(static_cast<const Derived&>(other)); }

  T& operator()(std::size_t row, std::size_t col)
  { return (*static_cast<Derived*>(this))(row, col); }

  const T& operator()(std::size_t row, std::size_t col) const
  { return (*static_cast<const Derived*>(this))(row, col); }
};

// Implementation template
template<typename T>
class RowMajorDenseMatrix : public Matrix<T, RowMajorDenseMatrix<T> > {
public:
  T& operator()(std::size_t row, std::size_t col) { ... }
  const T& operator()(std::size_t row, std::size_t col) const { ... }
  void assign(const RowMajorDenseMatrix<T>& other) { ... }
};
```

The most important aspect of the Barton & Nackman trick is that algorithms can be implemented in terms of the `Matrix<T, Derived>` template, which constrains their interface but does not negatively impact performance:

```
template<typename T, typename Derived>
  Matrix<T, Derived>
  operator*(const Matrix<T, Derived>& A, const Matrix<T, Derived>& B);
```

Without the Barton & Nackman trick, one would need to either provide separate `operator*` implementations for the different derived classes or would need to write a catch-all template[6]:

```
template<typename Matrix>
  Matrix operator*(const Matrix& A, const Matrix& B);
```

This proposal obsoletes the Barton & Nackman trick. The catch-all template could be restricted to require types that model a Matrix concept, and each implementation would provide a model definition for the Matrix concept. This solution provides stronger type checking than what can be done with Barton & Nackman, requires less effort for the developer, and does not rely on type casts.

[5] The Curiously Recurring Template Pattern also uses this construct.

[6] We now know that these templates can be restricted with `enable_if`, but Barton & Nackman found this solution nearly ten years prior!

References

[BN94] John Barton and Lee Nackman. *Scientific and Engineering C++*. Addison-Wesley, 1994.

[Dim04] Peter Dimov. Language support for restricted templates. Technical Report N1696=04-0136, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2004.

[ISOI98] International Standardization Organization (ISO). *ANSI/ISO Standard 14882, Programming Language C++*. 1 rue de Varembé, Case postale 56, CH-1211 Genève 20, Switzerland, 1998.

[JWL04] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Algorithm specialization and concept constrained genericity. In *Concepts: a Linguistic Foundation of Generic Programming*. Adobe Systems, April 2004.

[Mye95] Nathan Myers. A new and useful technique: "traits". *C++ Report*, 7(5):32–35, June 1995.

[SDR03a] Bjarne Stroustrup and Gabriel Dos Reis. Concepts – design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2003. http://www.open-std.org/jtc1/sc22/wg21.

[SDR03b] Bjarne Stroustrup and Gabriel Dos Reis. Concepts – syntax and composition. Technical Report N1536=03-0119, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2003. http://www.open-std.org/jtc1/sc22/wg21.

[Sie00] Jeremy Siek. *Boost Concept Check Library*. Boost, 2000. http://www.boost.org/libs/concept_check/.

[SL00] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[Str03] Bjarne Stroustrup. Concepts – a more abstract complement to type checking. Technical Report N1510=03-0093, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, October 2003. http://www.open-std.org/jtc1/sc22/wg21.