Concepts for C++0x
Revision 1

Douglas Gregor[1] Jeremy Siek[3] Jeremiah Willcock[1] Jaakko Järvi[2]
Ronald Garcia[1] Andrew Lumsdaine[1]

1. Indiana University, Open Systems Lab, Bloomington, IN 47405
2. Texas A&M University, Computer Science, College Station, TX 77843
3. Rice University, Department of Computer Science, Houston, TX 77005

Document number: N1849=05-0109
Revises document number: N1758=05-0018
Date: 2005-08-26
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Contents

1 Changes from N1758

This section summarizes the major changes this document makes to our previous proposal, N1758 [SGG+05].

- We have produced a prototype implementation, called ConceptGCC, that is described in more detail in N1848 [GS05b].
- We have added structural concepts, for which models can be implicitly generated.
- We have tightened the specification of concepts, including a summary of the grammar and more complete specification of semantics, including:
  - Name lookup in constrained templates
  - Partial ordering of templates based on **where** clauses
  - Verifying model correctness
- We have introduced several syntactic changes, to reduce the number of keywords (re)used and simplify the use of concepts.
- We have provided a discussion of the background research we have conducted on language support for generic programming, comparing the features of several other programming languages.

2 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.

N1758 [SGG+05] provides an overview of generic programming terminology that will be used throughout this proposal. A short review of this terminology follows. 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 discusses the implementation of a concept-enabled C++ standard library.

Terminology review The key terms of generic programming are:

- 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 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 [SGG+05, 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. Implementable in existing compilers.
7. No C++03 programs become ill-formed, unless those programs only work due to defects in C++03.
8. Ability 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.

3.1 Background research

This proposal is the result of extensive research on applying the ideas of generic programming in several programming languages, experimentation on language features for generic programming within research languages, and formally analyzing language features relevant to generic programming.

In a comparative study between several programming languages, we evaluated the level of support for generic programming in eight languages [GJL+03, GJL+05]. The study involved a partial implementation of the Boost Graph Library [SLL02, SLL01] in each of these languages. The goals of the study were to understand which language features are necessary to support generic programming; to understand the extent to which specific languages support generic programming; and to provide guidance for development of language support for generics. Tables 1 and 2 collect the results of the comparative study, listing the language features found crucial or useful for generic programming, and the level of support for each feature within the studied languages. The last row, and the two rightmost columns, are not based on the study. Regarding the chart in Table 1, we point out the following:

- The "ConceptC++" column reflects our understanding on how C++, enhanced with concepts as described in this proposal, would be evaluated according to the criteria used.
- The "$\mathcal{G}$" column describes the support for generic programming of the research language $\mathcal{G}$ [Sie05, SL05a]. This language implements the core features necessary for generic programming in a pure form, which has allowed us to model their semantics formally, enabling a thorough study of the language mechanisms for generic programming. The current proposal draws from the design of $\mathcal{G}$ and experimentation with the language, such as implementing the analog of the STL in $\mathcal{G}$ [SL05b].

	C++	SML	OCaml	Haskell	Eiffel	Java	C#	Cecil	$\mathcal{G}$	ConceptC++
Multi-type concepts	-	●	○	●*	○	○	○	◒	●	●
Multiple constraints	-	◒	◒	●	○†	●	●	●	●	●
Associated type access	●	●	◒	◒‡	◒	◒	◒	◒	●	●
Constraints on assoc. types	-	●	●	●	◒	◒	◒	●	●	●
Retroactive modeling	-	●	●	●	○	○	◒	●	●	●
Type aliases	●	●	●	●	○	○	○	○	●	●
Separate compilation	○	●	◒	●	●	●	●	◒	●	◒
Implicit arg. deduction	●	○	●	●	○	●	◒	◒	●	●
Concept-based overloading	◒	○	○	○	○	◒	◒	●	◒	●

*Using the multi-parameter type class extension to Haskell 98 [PJM97]. †Planned language additions. ‡We did not evaluate the recently proposed extensions to Haskell to support associated types [CKPM05, CKP05]

Table 1: The level of support for language features important for generic programming within several programming languages. A black circle indicates full support, a white circle indicates poor support, and a half-filled circle indicates partial support. The rating of "-" in the C++ column indicates that C++ does not explicitly support the feature, but one can still program as if the feature were supported due to the permissiveness of C++ templates. The table is based on the results of an experimental study reported in [GJL+03, GJL+05], with the addition of columns for $\mathcal{G}$ and ConceptC++and row for concept-based overloading.

- The last row with the additional criterion of *concept-based overloading*, was added to clarify additional capabilities that concepts can bring to C++. In C++, overloading, or dispatching, based on which concepts a type models can be arranged by *tag dispatching* or using the enable_if template [JWHL03]. The Standard library uses tag dispatching, for example, in the implementations of the advance and distance functions. With concepts, such overloading can be implemented directly and naturally, without resorting to trickery.

- ConceptC++ has partial support for separate compilation. The C++ compilation model of templates (instantiation) does not need to change with the introduction of concepts, but generic definitions can be type checked separately from their uses. There is a small caveat in type-checking, though. As specified currently, type-checking does not guarantee that certain kinds of overloads would not result in errors at instantiation time. These cases are known (see Section 4.4.2 and [JWL04]), and represent a fundamental tension between separate type checking and the ability to specialize algorithms using concept-based overloading. Note that the evaluations for separate compilation and concept-based overloading for $\mathcal{G}$ are reversed, compared to the evaluations for ConceptC++; the design choice made in language $\mathcal{G}$ is the opposite to what we propose for C++. $\mathcal{G}$ favors full separate type checking and compilation for concept-based overloading and algorithm specialization.

3.2 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. However, see N1848 [GS05b] for a discussion of the various alternative compilation models and how they can be implemented with concepts. 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 non-dependent template 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

Criterion	Definition
Multi-type concepts	Multiple types can be simultaneously constrained.
Multiple constraints	More than one constraint can be placed on a type parameter.
Associated type access	Types can be mapped to other types within the context of a generic function.
Constraints on associated types	Concepts may include constraints on associated types.
Retroactive modeling	Indicates the ability to add new modeling relationships after a type has been defined.
Type aliases	A mechanism for creating shorter names for types is provided.
Separate compilation	Generic functions can be type checked and compiled independent of calls to them.
Implicit argument deduction	Indicates that the arguments for the type parameters of a generic function can be deduced and do not need to be explicitly provided by the programmer.
Concept-based overloading	Generic functions can be overloaded on the concepts that their type parameters are required to model.

Table 2: Glossary of Evaluation Criteria

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;
}
```

We can rewrite this unsafe algorithm using concept constraints. By introducing requirements on the template parameters (renamed to InIter, OutIter, and Pred, respectively), both in the template header and the **where** clause, the compiler can verify that the template is correct *at definition time*. Here is the type-safe version of the template:

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

ConceptGCC produces the following diagnostic, indicating that there is no match for the < operator on input iterators. The second part of the error message indicates that the only non-built-in operator < known to the compiler is for the difference_type of the input iterator.

```
copy_if_bad.C: In function 'OutIter copy_if(InIter, InIter, OutIter, Pred)':
copy_if_bad.C:9: error: no match for 'operator<' in 'first < last'
<path>/include/c++/4.0.1/bits/concepts.h:150: note: candidates are:bool std::SignedIntegral<typename std::Iterator
AssociatedTypes<_Iter>::difference_type>::operator<(const typename std::IteratorAssociatedTypes<_Iter>
::difference_type&, const typename std::IteratorAssociatedTypes<_Iter>::difference_type&)
```

3.3 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. Rationale for other design decisions is provided in N1758 [SGG+05].

3.3.1 Named conformance vs. structural conformance

In general, there are two basic approaches to establishing that a set of types models a concept: structural and named conformance. Structural conformance relies only on the signatures within a concept, ignoring the name of the concept itself. With structural conformance, a set of types models a concept if all of the syntactic requirements of that concept are met; any semantics required by the concept are implied by the structure. Named conformance, on the other hand, means that the name of a concept is significant: two concepts with identical structure but with different names are considered different. With named conformance, a set of types models a concept only if the user has explicitly declared that the semantics of the concept are met; the syntactic requirements are checked when this declaration is made. Systems based on named conformance often allow the syntax to be adapted within the model declaration.

Both structural conformance and named conformance are important for the design of generic libraries. Some concepts are "purely structural", in the sense that they have little or no semantics and therefore should not require explicit model declarations. For these concepts, structural conformance is important. Other concepts, however, have many semantic requirements that users must consider before concluding that a set of types does model a concept. Particularly in the case where two concepts are structurally similar but semantically different (e.g., InputIterator and ForwardIterator), structural conformance can lead to run-time errors whereas named conformance would not [GS05a].

This proposal provides support for both named and structural conformance. Concepts use named conformance by default (because it is safer in general). However, so-called **struct concept**s use structural conformance, but still permit explicit model declarations.

N1782 [SD05] also supports both structural and named conformance. In that proposal, structural conformance is the default but the use of "negative asserts" can emulate named conformance.

3.3.2 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*). N1782 [SDR03a] presents 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], and each can emulate the other.

The pseudo-signature approach describes the syntax via a set of function declarations, as illustrated on the left side of Figure 1 (we describe the full syntax of concept definitions in Section 4.1). 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 convertible to type T and returns a value convertible to **bool**. Note that pseudo-signatures differ from the abstract signatures described by N1782 [SD05], because abstract signatures do not permit conversions of the argument and result types. Pseudo-signatures do permit these conversions.

The valid-expression approach describes the valid syntax by writing it directly. The right side of Figure 1 illustrates the description of Less Than Comparable using the syntax of N1782 [SD05].

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

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

Figure 1: Less Than Comparable concept expressed via pseudo-signatures (left) and valid expressions (right).

We have opted to use pseudo-signatures for several reasons. First, they match closely with the declarations of operations that fulfill these requirements, e.g., the **operator**< on an STL vector has essentially the same declaration as the pseudo-signature for **operator**< in Figure 1. Second, pseudo-signatures can appear both as requirements in concepts and as implementations of those requirements in models, simplifying the task of writing a complete model for a concept. Third, the implementation of pseudo-signatures matches so long as one can forward from the pseudo-signature to the corresponding implementation and back. Finally, pseudo-signatures are more precise than valid expressions, specifying precisely how arguments are passed and the exact return types, which is crucial for type-checking of templates.

3.3.3 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<typename 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 (e.g., 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[1]. Alternatively, associated types are looked up in the scope of the **where** clause. Thus, if ForwardIterator<X> is in the **where** clause, value_type will find ForwardIterator<X>::value_type.

An alternative to placing associated types in the concept definition is to make them nested types of one of the type parameters. For instance, N1782 places the associated types of an iterator inside the iterator type itself, e.g.,

```
concept ForwardIterator<class Iter> {
  Value_type Iter::value_type;
  SignedIntegral Iter::difference_type;
  // more requirements...
};
```

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

Then, if type X is a ForwardIterator, X::value_type refers to the value_type of the iterator. This formulation ties associated types to a particular type in the concept. While this may work well for single-parameter concepts, it can cause ambiguities and confusion. Consider the following concept, which states that the type F can be called with a parameter of type T1, and the identity function object:[2]

```
template<typename F, typename T1>
struct concept Callable1
{
  CopyConstructible F::result_type;
  F::result_type operator()(F&, const T1&);
};

struct identity_t
{
  template<typename T> T operator()(T t) { return t; }
};
```

There are models Callable1<identity_t, T> for every T that is CopyConstructible, and the type returned by **operator**() will always be T. Thus, F::result_type in the following function will be **int** when forward<identity_t, **int**> is instantiated and **float** when forward<identity_t, **float**> is instantiated, even though F=identity_t in both cases:

```
template<typename F, typename T1> where {Callable1<F, T1>}
F::result_type forward(F& f, const T1& t1)
{
  return f(t1);
}
```

That F::result_type can take on different types even when F is fixed to identity_t is somewhat surprising. It indicates that the result_type isn't really a property of F; rather, it's a property of the concept Callable1<F, T1>. This fact can cause some interesting ambiguities when the same nested type F::result_type can get different definitions from different concepts, as in the following example:

```
template<typename F, typename T1, typename T2>
where {Callable1<F, T1>, Callable1<F, T2>}
F::result_type forward(F& f, const T1& t1, const T2& t2)
{
  F::result_type r1 = f(t1);
  F::result_type r2 = f(t2);
}
```

In this example, F::result_type can refer to either the F::result_type from Callable1<F, T1> or F::result_type from Callable1<F, T2>, which may be different. With associated types as part of the concept, this ambiguity does not occur, because the associated type is part of the concept, not part of a specific type.

We have opted to express associated types as nested types within concepts, because it closely matches existing practice (traits), provides convenient access to associated types, and eliminates the problems of ambiguities.

3.3.4 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<InputIterator InputIterator1, InputIterator InputIterator2>
  where { InputIterator<InputIterator1>::value_type == InputIterator<InputIterator2>::value_type,
```

[2] We have invented this syntax just for this example. It does not reflect the proposed syntax.

```
        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.

Same-type constraints cannot be emulated with a SameType concept, because their use in type-checking templates requires nontrivial changes to a compiler. Section 4.4.3 describes same-type constraints and type equality issues in more detail; implementation details and strategies are discussed in a separate document [GS05b].

4 Proposed language features

This proposal introduces three new kinds of entities into the C++ language: concepts, models, and **where** clauses. It also changes the nature of type-checking in templates, by introducing non-dependent template parameters for which all type checking occurs when the template is parsed. We will cover the syntactic entities (and their semantics) first, then discuss type-checking of constrained templates, and finish with potential extensions to these features.

4.1 Concepts

template-declaration:
`template` < *template-argument-list* > `struct`$_{opt}$ `concept` *identifier concept-definition*$_{opt}$;

concept-definition:
refinement-clause$_{opt}$ *concept-body*

concept-body:
{ *requirement-specification*$_{opt}$ }

requirement-specification:
pseudo-signature-req requirement-specification$_{opt}$
associated-type-req requirement-specification$_{opt}$
nested-req requirement-specification$_{opt}$

Concepts are namespace-level entities that bundle together a set of requirements. When the **struct** keyword precedes **concept**, the concept is a structural concept. The syntax of concepts closely mimics class templates. Concepts consist of an (optional) refinement clause and three kinds of members: pseudo-signatures, associated types, and nested requirements.

The following code defines a concept named ForwardIterator with a single template parameter, Iter:

```
template<typename Iter>
concept ForwardIterator { ... };
```

4.1.1 Refinements

refinement-clause:
: *refinement-specifier-list*

refinement-specifier-list:
refinement-specifier
refinement-specifier , *refinement-specifier-list*

refinement-specifier:
model-id

The refinement clause contains a list of *refinement-specifier*s that indicate which concepts are refined by the concept being defined and how parameter substitutions affect this refinement relationship. A *model-id* is a *template-id* whose

template-name refers to a concept. Concept refinements shall not be recursive. The intuition is that a refinement adds the requirements of the refined concept to the refining concept. Thus, the requirements in the refining concept are a superset of those in the refined concept, and any set of types that models the refining concept also models the refined concept.

The following code defines a concept BidirectionalIterator that refines ForwardIterator:

```
template<typename Iter>
concept BidirectionalIterator : ForwardIterator<Iter> { ... };
```

In addition to organizing and aggregating requirements, refinements affect the implicit generation of models (Section 4.2.3) and partial ordering based on **where** clauses (Section 4.3).

4.1.2 Pseudo-signatures

pseudo-signature-req:
    simple-declaration
    function-definition
    template-declaration

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 or definition. 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 4.2 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<typename T>
concept EqualityComparable {
  bool operator==(const T&, const T&);
  bool operator!=(const T& x, const T& y) { return !(x == y); }
};
```

Operators should always be written like free functions within a concept, even if those operators may only be defined as members within a class. For instance, the Convertible concept is written as a free function, even though conversions can be built-in, performed through constructors, or written as member operators in a class.

```
template<typename T, typename U>
struct concept Convertible {
  operator U(const T&);
};
```

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<typename 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<typename T>
concept DefaultConstructible {
```

```
  T::T();
};
```

A concept may require a function template via a pseudo-signature template. The following is an example of a pseudo-signature template:

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

4.1.3 Associated types

associated-type-req:
    typename identifier ;
    typename identifier = *type-specifier* ;

A concept may require that a model provide a type definition for a particular type name via a **typedef**. These requirements are called *associated type* requirements. For example, the following Graph concept requires that a model declaration for Graph specify a type for edge and vertex.

```
template<typename 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<typename T>
concept A {
  typename s;
};
template<typename 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 a type definition for that type, 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<typename 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;
  ...
};
```

4.1.4 Nested requirements

nested-req:
`where` *requirement-list* `;`

Nested requirements are additional requirements that a concept places on its type parameters, associated types, etc. For instance, a concept may require certain types to model another concept with a nested **where** clause that names the required *model-id*. The Container concept requires that the associated type iterator satisfy the requirements of InputIterator, which is written as:

```
template<typename X>
concept Container {
  ...
  typename iterator;
  where InputIterator<iterator>;
  ...
};
```

Nested requirements may contain the same kinds of requirements as a **where** clause; see Section 4.3.

4.2 Models

template-declaration:
`template` < *template-argument-list*$_{opt}$ > *where-clause*$_{opt}$ `concept` *model-id model-body*$_{opt}$ `;`

model-body:
{ *model-member-specification*$_{opt}$ }

model-member-specification:
pseudo-signature model-member-specification$_{opt}$
associated-type model-member-specification$_{opt}$

A model definition establishes that a set of types meets the requirements of a concept. Syntactically, a model is like a (partial) specialization of a concept. Consider the following example:

```
class student_record {
public:
  string id;
  string name;
  string address;
};

template<>
concept 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 4.1.2) 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**==.

4.2.1 Verifying model correctness

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 is required.

Pseudo-signatures

pseudo-signature:
    pseudo-signature-req

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<typename T>
concept A {
  typename s;
  s foo(T);
};

template<>
concept 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.

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

   Similarly, the pseudo-signature may be declared inside the model definition but defined outside:

   ```
   template<>
   concept A<int> {
     typedef char s;
     char foo(int x);
   };

   char A<int>::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<typename U>
concept B : A<T> { };

template<>
concept 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 declaration or definition in the model body or by a model of a refinement), then the pseudo-signature shall be implicitly defined. For details, see Section 4.2.2.

Associated types

associated-type:
`typedef` *type-specifier identifier* **;**

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<typename T>
concept A {
  typename t;
};

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

template<>
concept B<float> { }
```

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

```
template<>
concept 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**>.

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

template<>
concept C<float> { };
```

Nested requirements Once a model has been defined, the nested requirements of the corresponding concept (and its refined concepts) must be verified to be true. This verification is the same as verifying that the **where** clause of a template is satisfied (Sections 4.3.1–4.3.3).

For instance, a nested model requirement must be satisfied by a previous model definition. Consider the following example.

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

template<typename U>
concept B {
  where A<U>;
};

template<>
concept A<int> { };

template<>
concept 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 **where** A<U>;, are replaced by the arguments and type definitions in the model, so U is replaced by **int**.

4.2.2 Implicit model member definitions

Since every model must contain precisely the same members (pseudo-signatures, associated types, etc.) as the concept it is associated with, members not explicitly defined by the user will be defined implicitly by the implementation. This behavior is very similar to the handling of implicitly-defined default constructors, copy constructors, and assignment operators.

This section describes how the implicitly-generated model members shall be defined.

Pseudo-signatures Implicitly-defined pseudo-signatures are implemented by creating a forwarding function whose signature exactly matches the pseudo-signature. The body of this function consists of a function call to the result of the function lookup (for function pseudo-signatures) or uses the operator in the least restrictive manner (for operator pseudo-signatures). The normal C++ lookup rules are applied when compiling this function. In the following example, the requirement for **operator**+ is satisfied by X::**operator**+.

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

template<>
concept C<X> { };
```

The definition of the implicitly-generated pseudo-signature looks like this:

```
bool C<X>::operator+(const X& x, const Y& y) {
  return x + y;
}
```

When compiling this function, the + operation resolves to X::**operator**+, and the **int** result of X::**operator**+ is implicitly converted to the **bool** result type required by the concept. Free functions are translated to unqualified calls. For instance, consider the following example for the concept Swappable:

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

template<typename T> where { CopyConstructible<T>, Assignable<T> }
void swap(T& x, T& y) { // #1
  T tmp(x);
  x = y;
  y = x;
}

template<>
concept Swappable<int> { };

// implicitly−generated!
void Swappable<int>::swap(int& x, int& y) {
  swap(x, y); // ignores Swappable<int>::swap but finds #1
}
```

For a more precise formulation of implementations for these synthesized pseudo-signatures, see our paper "Implementing Concepts" [GS05b].

If the implicitly-generated definition fails to type-check and the pseudo-signature requirement in the concept contains a default implementation, that default implementation will be used instead. The example at the beginning of Section 4.1.2, with the model EqualityComparable<student_record>, demonstrates the use of default implementations in concepts.

If the implicitly-generated definition fails to type-check and there is no default implementation, the compiler shall produce a diagnostic indicating that the model is invalid.

Associated types If a model definition does not provide a **typedef** for an associated type, the default type value will be used if provided in the concept. For instance, the following model of InputIterator retrieves the difference_type type from MyIter:

```
template<typename Iter>
concept InputIterator {
  typename difference_type = Iter::difference_type;
};

struct MyIter {
  typedef int difference_type;
};

template<>
concept InputIterator<MyIter> { };
```

4.2.3 Refinements and 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<typename T>
concept A {
  void foo();
};

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

template<>
concept 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**>.

Models for refined concepts will be defined even for model templates. These implicitly defined models will have the same template parameters, **where** clauses, and template arguments as the model originally defined by the user. For instance, if ForwardIterator refines InputIterator, the following model of ForwardIterator will result in a similar model of InputIterator:

```
template<typename T>
concept ForwardIterator<T*> {
  typedef T value_type;
  typedef std::ptrdiff_t difference_type;
  typedef const T& reference;
  typedef const T* pointer;
};

// Implicitly generated, unless it already exists...
template<typename T>
concept InputIterator<T*> {
  typedef T value_type;
  typedef std::ptrdiff_t difference_type;
  typedef const T& reference;
```

```
  typedef const T* pointer;
};
```

If the template parameters of an implicitly defined model are not all deducible from the template arguments of that model, the model shall not be defined. For instance, consider the following concept and model:

```
template<typename V, typename S>
concept VectorSpace : AbelianGroup<V>, Field<S> { ... };

template<typename T, typename U>
  where { Convertible<U, T> }
concept VectorSpace<std::complex<T>, U> { ... };
```

Within the above rule, the implementation would attempt to implicitly define:

```
template<typename T, typename U>
  where { Convertible<U, T> }
concept AbelianGroup<std::complex<T> > { ... };

template<typename T, typename U>
  where { Convertible<U, T> }
concept Field<U> { ... };
```

However, both of these are invalid partial specializations (and, hence, invalid model declarations), because U cannot be deduced in the first definition and T cannot be deduced in the second definition.

4.2.4 Model identifiers

model-id:
    template-id

A concept name followed by a list template arguments is a model identifier (*model-id*). 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(); }
```

Lookup into a *model-id* is only well-formed when there exists a model for that *model-id*. The method used to determine if a model exists is described in Section 4.3.1.

4.2.5 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, respectively.

```
template<typename T>
concept MutableRandomAccessIterator<T*> {
  typedef T value_type;
  typedef T& reference;
  typedef T* pointer;
  typedef ptrdiff_t difference_type;
};

template<typename T>
```

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

In the upcoming Section 4.3 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<typename T, typename Alloc> where { EqualityComparable<T> }
concept EqualityComparable< vector<T, Alloc> > { };
```

This model template states that a vector is a model of EqualityComparable if the value type T is EqualityComparable.

4.2.6 Friend models

member-declaration:
`friend concept` *model-id* `;`
`template <` *template-argument-list* `> friend concept` *concept-name* `;`

Models can be friends of a class, permitting the definition of those models to access **private** or **protected** members of the class. For instance, we can rewrite the previous student_record example with all its members **private**:

```
class student_record {
private:
  string id;
  string name;
  string address;

  friend concept EqualityComparable<student_record>;
};

template<>
concept EqualityComparable<student_record> {
  bool operator==(const student_record& a, const student_record& b)
  { return a.id == b.id; }
};
```

4.3 Where clauses

template-declaration:
`export`$_{opt}$ `template` < *template-parameter-list* > *where-clause*$_{opt}$ *declaration*

member-declaration:
where-clause member-declaration

where-clause:
`where` { *requirement-list*$_{opt}$ }

type-parameter:
class `!`$_{opt}$ *identifier*$_{opt}$
class `!`$_{opt}$ *identifier*$_{opt}$ = *type-id*
typename `!`$_{opt}$ *identifier*$_{opt}$
typename `!`$_{opt}$ *identifier*$_{opt}$ = *type-id*
template < *template-parameter-list*$_{opt}$ > `class` `!`$_{opt}$ *identifier*$_{opt}$
template < *template-parameter-list*$_{opt}$ > `class` `!`$_{opt}$ *identifier*$_{opt}$ = `id-expression`

requirement-list:
requirement , *requirement-list*

requirement:
model-requirement
same-type-requirement
ice-requirement

This proposal introduces constraints on templates in the form of a **where** clause. The syntax of template declarations (and definitions) is extended to include a **where** clause, which consists of a set of requirements. Any template that contains a **where** clause is called a *constrained template*. The following is a simple example:

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

where clauses can also be placed on the members of class templates. For instance, std::list<T>::sort can only be applied when T is a model of LessThanComparable:

```
template<typename T> where { CopyConstructible<T> }
class list {
public:
  where { LessThanComparable<T> } void sort();
};
```

The requirements in a **where** clause play two roles in type checking:

1. When a template identifier is used, such as vector<**int**>, all of the requirements in the template's **where** clause must be satisfied, otherwise the program is ill-formed.

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

The first of these roles is discussed in this section, which describes how checking for each of the kinds of requirements shall occur. Failure to satisfy the requirements of a **where** clause means that the template cannot be used (e.g., called or instantiated). For instance, consider the following simple 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. ConceptGCC emits the following message:

```
sort.C:7: error: no matching function for call to 'sort(std::_List_iterator<int>, std::_List_iterator<int>)'
<path>: note: candidates are: void std::sort(_Iter, _Iter) [with _Iter = std::_List_iterator<int>] <where clause>
sort.C:7: note: unsatisfied model requirement 'std::MutableRandomAccessIterator<std::_List_iterator<int> >'
```

Discussion of the second role of **where** clauses, as constraints that add assumptions to the context of the compilation, is deferred to Section 4.4.

4.3.1 Model requirements

model-requirement:
    model-id

A model requirement is a *model-id*, which names a concept and provides it with template arguments. A model requirement is only satisfied if there exists a model declaration that best matches the template 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 (as is done with class and function templates). Partial ordering of model templates occurs in the same manner as partial ordering of class templates; this proposal extends the partial ordering rules to also consider **where** clauses, described in Section 4.3.5. Consider the following example:

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

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

template<typename T>
concept A<T*> { };

template<>
concept 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.

Model requirements in templates Model requirements can also be satisfied by model requirements expressed or implied in the **where** clause of an enclosing scope. For instance, the call to lower_bound inside binary_search is valid because binary_search's **where** clause contains all of the models required by lower_bound:

```
template<typename Iter>
where { BidirectionalIterator<Iter>, LessThanComparable<value_type> }
Iter lower_bound(Iter first, Iter last, value_type value);

template<typename Iter>
where { BidirectionalIterator<Iter>, LessThanComparable<value_type> }
bool binary_search(Iter first, Iter last, value_type value) {
  Iter result = lower_bound(first, last, value);
  // ...
}
```

Model requirements for structural concepts If a model requirement cannot be satisfied with any existing model and the corresponding concept is a structural concept, the model requirement can be satisfied by a successful *structural match*. A structural match occurs when a set of types fulfills all of the syntactic requirements of a structural concept.

Structural matches are attempted only when no other models exist for a *model-id*. When a structural match fails, the model requirement is unsatisfied but the program is not necessarily ill-formed [3]. For instance, consider the following example:

```
template<typename T, typename U>
struct concept Convertible {
  operator U(const T&);
};

template<typename T> where { Convertible<T, int> }
int f(const T& t);

void g(char ch) {
  f(ch); // ok
}
```

There is no explicit model declaration Convertible<**char**, **int**>, so a structural match is attempted. It succeeds, so the program is well-formed.

Structural matches are performed as if an empty, non-templated model definition were created at the point where the model is required. For instance, the model definition generated for the above example is:

```
template<> concept Convertible<char, int> { };
```

4.3.2 Same-type requirements

same-type-requirement:
    type-specifier == *type-specifier*

A same-type constraint is satisfied if the two types are equivalent. Consider the following example:

```
template<typename T, typename 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
```

Within a template, determining if a same-type constraint is satisfied may require comparisons based on same-type constraints expressed in or implied by the **where** clause of an enclosing scope. See Section 4.4.3 for more details on type equivalence with same-type constraints.

4.3.3 Integral constant expression requirements

ice-requirement:
    assignment-expression

An integral constant expression requirement is satisfied if the integral constant expression, when converted to a **bool**, evaluates **true**. For instance, the following class template accepts only odd integers: any attempt to provide it with an even integer will result in a diagnostic:

[3] Readers concerned about the implementability of this feature may wish to read section 3.6 of N1848 [GSW+05].

```
template<int N> where { N % 2 }
struct only_odd { };

only_odd<5> five; // OK
only_odd<6> seven; // Error: N % 2 does not evaluate true
```

4.3.4 Constraint propagation

It is often the case that certain requirements on template parameters are apparent from the declaration of a constrained template, even if they are not explicitly stated. These requirements (constraints) are implicitly added to the **where** clause of the template through the process of *constraint propagation*. Constraint propagation must generate the smallest set of implicit constraints that guarantee that the declaration of a template will always instantiate properly.

One immediate example of an implicit constraint is that pass-by-value parameters and return types are assumed to model CopyConstructible. Thus, the following template is well formed:

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

The constraint propagation rule has some interesting implications. Take the following definition of vector:

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

In the following declaration, the template includes an implicit constraint that T cannot be a reference type:

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

4.3.5 Partial ordering with where clauses

Function and class templates can be partially ordered based on their function arguments and template arguments, using the rules in 14.5.5.2 and 14.5.4.2 of the C++ standard, respectively. This proposal extends this ordering when two templates are considered identical after removing the **where** clauses and modulo the names of template parameters. In this case, the templates are partially ordered based on the requirements in the **where** clauses.

The following examples illustrate intuitively how this partial ordering should work. In the following example, the second f should be chosen because B refines A.

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

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

template<> concept 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<typename T> concept A { };
template<typename T> concept C { };

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

template<> concept A<int>;
template<> concept 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<typename T> concept A { };
template<typename T> concept B : A<T> { };
template<typename T> concept C { };

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

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

Given two templates T_1 and T_2 that are equivalent modulo template parameter names and **where** clauses, use the following procedure to determine a partial ordering between the templates.

1. Introduce the requirements from the **where** clause of T_1 into a new environment.
2. Check each of the requirements in the **where** clause of T_2 to determine if they are satisfied in the new environment. If so, T_1 is at least as specialized as T_2.
3. Repeat the process with a new environment, to determine if T_2 is at least as specialized as T_1.
4. If T_1 is at least as specialized as T_2, but T_2 is not at least as specialized as T_1, then T_1 is the more specialized template. Similarly, we can determine if T_2 is more specialized than T_1.

4.3.6 Syntactic shortcut for single-parameter concepts

template-parameter:
    concept-name !$_{opt}$ *identifier*
    concept-name !$_{opt}$ *identifier* = *assignment-expression*

type-parameter:
    concept-name !$_{opt}$ *identifier* = *type-id*
    concept-name !$_{opt}$ *identifier* = *id-expression*

It is common for many concepts in a program to have only a single template parameter, typically a template type parameter. To make these concepts more easy to use in a template, we provide a shortcut (also in N1782 [SD05]) wherein the concept name may be written as the "type" of the template parameter, instead of **typename** or **class**. For instance, here is a declaration of advance() that uses this shortcut:

```
template<InputIterator Iter> void advance(Iter& x, difference_type n);
```

Using this shortcut is identical to writing out the template with a **where** clause, e.g.,

```
template<typename Iter> where { InputIterator<Iter> }
void advance(Iter& x, difference_type n);
```

4.4 Type checking templates

The requirements placed on template parameters by a **where** clause have two roles. When using a constrained template, the requirements of the **where** clause must be satisfied by the user. However, those same requirements are also taken as assumptions against which the body of the template can be fully type-checked. This section focuses on the latter role.

4.4.1 Non-dependent template parameters

This proposal introduces non-dependent template parameters, which, unlike normal template parameters, do not make expressions based on them dependent. In essence, a non-dependent template parameter acts more like a regular, non-dependent type (**class** X, **int**, **char**$*$) than a template parameter. Name lookup and type-checking for non-dependent types and expressions occurs when a template is initially defined, whereas lookup and checking for dependent types and expressions is delayed until instantiation time. With non-dependent template parameters, all type-checking and name lookup can occur at template definition time, so that errors can be detected prior to instantiation. Modulo certain ambiguities and problems with incorrectly specified specializations, a well-formed template that uses only non-dependent template parameters is guaranteed to instantiate properly.

When any requirements or a **where** clause is provided for a template, all template parameters are considered non-dependent unless specifically marked as "dependent." An empty **where** clause also suffices to make template parameters non-dependent. Let us start with a generic for_each algorithm and introduce an empty **where** clause, to enable type-checking:

```
template<typename Iter, typename F> where {}
F for_each(Iter first, Iter last, F f)
{
  while (first != last) {
    f(*first);
    ++first;
  }
  return f;
}
```

This program is ill-formed, because we have stated that there are no requirements on the template parameters Iter and F, but we are using many operators. For this code snippet, ConceptGCC produces the following output:

```
for_each.C: In function 'F for_each(Iter, Iter, F)':
for_each.C:6: error: no match for 'operator!=' in 'first != last'
for_each.C:7: error: no match for 'operator*' in '*first'
for_each.C:7: error: 'f' cannot be used as a function
for_each.C:8: error: no match for 'operator++' in '++first'
for_each.C:10: error: 'F' has no copy constructor
```

To eliminate the error messages related to the iterator operations !=, $*$, and ++, we need to state that the Iter type is actually an InputIterator. We do so using the std::InputIterator concept, which will presumably be a part of a concept-enabled Standard Library (as it is in ConceptGCC's standard library implementation). Doing so results in the following code:

```
template<std::InputIterator Iter, typename F> where {}
F for_each(Iter first, Iter last, F f)
{
  while (first != last) {
    f(*first);
    ++first;
```

```
  }
  return f;
}
```

Is this code now correct? Attempting to compile it with ConceptGCC produces the following error messages:

```
for_each.C: In function 'F for_each(Iter, Iter, F)':
for_each.C:7: error: 'f' cannot be used as a function
for_each.C:10: error: 'F' has no copy constructor
```

F needs to be some type that is CopyConstructible so that f can be returned[4]. F must also be a function pointer or function object (or anything else "Callable"). With these constraints added, the definition of for_each becomes:

```
template<std::InputIterator Iter, std::CopyConstructible F>
  where { std::Callable1<F, reference> }
F for_each(Iter first, Iter last, F f)
{
  while (first != last) {
    f(*first);
    ++first;
  }
  return f;
}
```

The reference type refers to std::InputIterator<Iter>::reference, the return type of **operator***. This final definition of for_each is now well-formed and compiling it with ConceptGCC produces no errors.

There are many template libraries in existence now that would benefit from the introduction of concepts. However, introducing concepts into an existing library is not a trivial task, as we have found while updating the GNU C++ standard library implementation. To ease the transition, it is possible to mark template parameters as "dependent" even when we have placed requirements on them via a **where** clause. To do this, we place a ! in front of the name of the template parameter:

```
template<std::InputIterator !Iter, std::CopyConstructible !F>
  where { std::Callable1<F, reference> }
F for_each(Iter first, Iter last, F f) {
  // Do some metaprogramming and perhaps parallelize the loop
  return f;
}
```

With the ! operators in place, the template parameters are dependent, so the body of for_each is largely unchecked. Thus, we can use whatever template metaprogramming we want, even if it would be hard or impossible to do in a fully type-checked template. The type-checking for this for_each is essentially one-sided: users must meet the requirements of the **where** clause when calling for_each, but there are no requirements on the implementor of for_each. Practically speaking, this allows template library authors to introduce concepts gradually, providing better error messages and checking for users initially while evolving the implementation to a completely type-checked, safe version.

4.4.2 Name lookup

This proposal introduces two additional rules that affect name lookup in templates. The first rule involves introducing the ability to perform lookups in the model requirements in a **where** clause, so that pseudo-signatures and associated types may be used unqualified within constrained templates. The second rule involves the lookup of unqualified names for function calls in constrained templates.

[4] This is actually due to a bug in ConceptGCC, which does not support constraint propagation. However, explicitly writing the constraints is not harmful.

Lookup in where clauses The **where** clause introduces names into the scope of the template definition. For each model requirement, the pseudo-signatures from the corresponding concept, its refinements, and all nested requirements (including their refinements!) are in scope. Additionally, the associated types from the concept and its refinements (but *not* nested requirements) are in scope. The scope of these requirements is the same as the scope of the template parameter list.

Pseudo-signatures Pseudo-signatures of model requirements introduce function declarations into the template parameter list scope. For example, consider the definition of swap().

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

The model 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. Similarly, the model requirement CopyConstructible<T> brings the declaration

```
T::T(const T&);
```

into scope, allowing the variable initialization T tmp = a to type check.

Associated types Associated type requirements introduce type names but not the actual type bindings. The following example demonstrates how a **where** clause brings associated types into scope:

```
template<Container C, typename F> where { Callable1<F, 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 unless the types can be proven equivalent. Consider the following example:

```
template<typename 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<typename C1, typename 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.

Non-dependent names When name lookup in a template resolves to something outside the scope of the template itself (or its **where** clause), the name may reference an overloaded function. The handling of this name depends on whether any arguments to the function are dependent or not. If any are dependent, the call expression is dependent and the overload set will be augmented by functions found through argument-dependent lookup (ADL) at instantiation time. If they are not dependent, this overload set is complete and the compiler will perform overload resolution when a single function is needed.

The introduction of constrained templates does not change this aspect of name lookup. Consider the following example, in which lower_bound makes an unqualified call to advance:

```
template<InputIterator Iter>
void advance(Iter& x, difference_type n); // #1

template<BidirectionalIterator Iter>
void advance(Iter& x, difference_type n); // #2

template<BidirectionalIterator Iter>
Iter lower_bound(Iter first, Iter last, const value_type& value)
{
  difference_type n = distance(first, last);
  Iter mid = first;
  advance(mid, n/2);
  // ...
}

template<RandomAccessIterator Iter>
void advance(Iter& x, difference_type n); // #3
```

Name lookup on advance finds the two advance overloads, with different concept requirements. Since mid and n/2 are non-dependent expressions, overload resolution is performed immediately. Both #1 and #2 match, but #2 is more specialized so it will be selected. If #3 was visible (i.e., declared prior to lower_bound), it would have been rejected because Iter is not necessarily a RandomAccessIterator.

When an call expression is non-dependent but contains (non-dependent) template parameters, the overload set returned by name lookup will *not* be augmented by functions found through ADL at instantiation time. Instead, it will be augmented with the *implied specializations*[5] of advance, i.e., any other declarations of advance in the same namespace that are (1) identical modulo template parameter names and **where** clauses and (2) more specialized, based on the **where** clause. Definition #3 fits the criteria of an implied specialization, so it will be found and added to the overload set *at instantiation time*. Continuing the example above, we define an iterator type my_iter that has its own advance:

```
namespace other {
  struct my_iter { typedef int difference_type; ... };

  void advance(my_iter& x, int n); // #4
}

namespace std {
  template<> concept RandomAccessIterator<other::my_iter> {};
}
```

When lower_bound<other::my_iter> is instantiated, which declaration of advance will the call in lower_bound resolve to? #1 and #3 are implied specializations, so they will enter the overload set. #4 will not be found because ADL is not employed for non-dependent expressions at instantiation time. Since other::my_iter is a model of RandomAccessIterator, and #3 is the most specialized function in the overload set, it will be used.

[5] This notion comes from N1782 [SD05].

These name lookup rules permit specializations of algorithms to occur and be used by other, type-safe generic algorithms without the problems caused by argument-dependent lookup. For instance, the resolution of Library DR 225 ("std:: algorithms use of other unqualified algorithms") is automatic when the algorithms in namespace std are constrained templates.

Lookup in uninstantiated templates Type checking the definition of a constrained template is performed without knowledge of the bindings for the template parameters. Thus it is impossible to instantiate templates according to template identifiers that appear in the body. Instead, the best matching specialization will be chosen (and used for type checking purposes) with the information at hand.

Consider the following well-formed C++03 program. The function template f() is instantiated with U=**int** and the template identifier foo<U> resolves to the specialization foo<**int**>.

```
template<typename T> class foo { };
template<> class foo<int> { void bar(); };
template<typename U> void f() { foo<U>::bar(); }
int main() { f<int>(); }
```

If f() is changed to a constrained template, the body is checked without knowledge of what U will be. In this case, foo<U> will refer to the primary template, which does not have a member bar(), and a diagnostic will be issued. The rationale behind these type checking rules is that if the template f() is truly generic it should work with any type bound to parameter U, not just **int**.

```
template<typename U> where { EqualityComparable<U> }
void f() { foo<U>::bar(); } // error
```

It is possible for a specialization to be chosen. For example, if there was a specialization of foo for T*, and a use of the template identifier foo<U*>.

Once f is instantiated, template instantiation will occur inside the definition of f as usual. This may result in different specializations than were used during type checking, and thus may result in compilation errors that were missed by the initial type checking. For example, the definition of g() in the program below will type check based on the primary template for foo but will fail when g is instantiated and the specialization foo<**int**> is selected.

```
template<typename T> class foo { int bar(); }
template<> class foo<int> { void bar(); }
template<typename U> int g() { return foo<U>::bar(); }
int main() { return g<int>(); } // instantiation−time diagnostic
```

4.4.3 Type equivalence

A same-type constraint expresses the requirement that two type expressions denote the same type. For instance, in the following example we state the relationship between the pointer, reference, and value_type associated types in a Mutable Forward Iterator:

```
template<typename X>
concept MutableForwardIterator {
  typename value_type;
  typename reference;
  typename pointer;
  where reference == value_type&;
  where pointer == value_type*;
};
```

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. Same-type constraints introduce "deep" equivalence between the two classes. For instance, S == T implies S* == T*, vector<T> == vector<U>,
vector<T>::value_type == vector<U>::value_type, etc. 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. Similarly, the same-type constraint S* == T* implies S == T, and everything implied by it.

4.5 Extensions

This section documents potential extensions to the features we are proposing. The list of extensions is not exhaustive, but contains only features that we know would be useful but we do not require because either (1) we are not sure they are implementable or (2) we have not had the time to adequately specify them.

4.5.1 Partial specialization of function templates

The implied specializations described in Section 4.4.2 are similar in spirit to partial specializations of function templates. They use the same name as the function template, have equivalent declarations modulo parameter names, and have **where** clauses that are more restrictive than the original template. If partial specializations of function templates were allowed, the partial specializations of the functions in the overload sets should augment the overload set at instantiation time (as the implied specializations do now).

Consider the following declarations of copy, one of which uses arbitrary input and output iterators and the other uses pointers:

```
template<InputIterator InIter, typename OutIter>
  where { OutputIterator<OutIter, InputIterator<InIter>::value_type> }
  OutIter copy(InIter first, InIter last, OutIter out); // #1

template<typename T>
  T* copy(T* first, T* last, T* out); // #2
```

If a generic function taking input and output iterators contained a call to copy, only #1 would enter the overload set because #2 would not match to arbitrary input and output iterators. Thus, we miss out on the opportunity to pick up a better match. However, if #2 was written as a partial specialization of a function, it would be picked up when the call to copy is instantiated:

```
template<typename T>
  T* copy<T*, T*>(T* first, T* last, T* out); // #3
```

We note that the behavior of #3 can be emulated with the current proposal, but leave the formulation as an exercise for the reader.

4.5.2 Associated values

Concepts permit associated type requirements to be defined, but there is no analogue for "associated values", i.e., integral constant expressions that are part of the concept. Associated value requirements would contain the type of the associated value, but the value itself would be unknown until instantiation time.

This extension was proposed by Daniel Krügler on `comp.std.c++`. We do not yet have a favored syntax for associated values nor have we attempted to implement them, but we do not believe that there is anything fundamentally difficult in their implementation.

4.5.3 Nested class template requirements

Concepts may include pseudo-signature templates that require their models to have templated operations. One could also consider permitting concepts to require nested class templates, and then place requirements on the members of those class templates. For instance, we could specify an Allocator concept similar to the one in the standard library with such an extension:

```
template<typename Alloc>
concept Allocator {
  template<typename T>
```

```
  struct Alloc::rebind {
    typename other;
    where Allocator<other>;
  };
  // ...
};
```

In this example, we want to say that every model of Allocator has a member template **struct** named rebind that contains a type named other that itself models the Allocator concept.

The ability to specify these kinds of requirements may make it possible to express type-safe template metaprograms using concepts. However, at this time we are not sure that such an extension is implementable, and it is an active research area.

4.5.4 Remote specializations and models

Specializations (and models) need to be written in the same namespace as the primary template (or concept). This restriction is rather inconvenient when the types used in the specialization come from a different namespace. We could allow "remote" specializations that can be written in any namespace. These specializations would have the same semantics as existing specializations, except that name lookups within the specializations themselves would reflect their point of declaration. For instance:

```
namespace std {
  template<typename X>
  concept InputIterator { ... };
}

namespace boost {
  template<typename Func>
  class counting_iterator {
    // ...
  };

  template<typename Func>
  concept std::InputIterator<counting_iterator<Func> > {
    // ...
  };
}
```

4.5.5 Exporting defaulted requirements

Default implementations of pseudo-signatures allow generic functions to use a richer syntax than is required by the concepts themselves. For instance, the LessThanComparable concept only requires **operator**< to be defined, but any function requiring LessThanComparable<T> may use operators <, >, <=, and >=:

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

We could invent a syntax that couples remote models (Section 4.5.4) with the idea of "exporting" the default implementations for a class when it is defined. This might be similar to the way in which **friend** functions defined in a class template are exposed. For instance, we might define a class X that itself contains only **operator**<, but by exporting default requirements we get >, <=, and >=:

```
struct X {
  friend bool operator<(const X& x, const X& y);
  export concept std::LessThanComparable<X>;
};

bool equiv(X x, X y) {
  return !(x < y) && !(y < x); // Ok!
}
```

This feature would make it possible to get the benefits of the Boost Operators library [AS02] from concepts, so that model declarations can actually reduce the amount of typing required to define a type.

5 Impact

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

5.1 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 some of 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 careful use of structural concepts, and by proper compiler diagnostics that describe what models are required. For instance, compiling the entire ConceptGCC testsuite (for both GCC and its standard library) with the concept-enabled C++ standard library required the introduction of only one explicit model declaration; that declaration could be avoided if GCC supported the decltype proposal [JS03].

5.2 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 and prototyped in ConceptGCC:

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 EqualityComparable when T models EqualityComparable.
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 RandomAccessIterator. We can 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; see Section A.

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.

5.3 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. The impact that this proposal will have on compilers, and a discussion of the techniques we used to implement concepts for ConceptGCC and the $\mathcal{G}$ compiler, are provided in a separate document [GS05b].

6 Acknowledgments

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 and NSF grant EIA-0131354. The third author was supported by a Department of Energy High Performance Computer Science Fellowship.

A Example: Standard library concepts and declarations

In addition to adding concept support to the GNU C++ compiler to create ConceptGCC, we also updated a large part of the GNU C++ standard library implementation, `libstdc++`, to use concepts. To demonstrate the syntax of our proposal and that it is sufficient to express the type requirements of the C++ standard library, we present some examples from this updated standard library. 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). This is a structural concept, so any type with a $<$ operator similar to the given signature will model this concept, and will have the other operators defined automatically.

Table 29, Less Than Comparable requirements [Int98]

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<typename T>
struct concept LessThanComparable {
  bool operator<(const T&, const T&);
  bool operator<=(const T& x, const T& y)
    { return !(y < x); }
  bool operator> (const T& x, const T& y)
    { return y < x; }
  bool operator>=(const T& x, const T& y)
    { return !(x < y); }
};
```

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

Table 30, Copy Constructible requirements [Int98]

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<typename T>
struct 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 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<typename Ptrlike, typename Value>
struct concept Arrowable
{
  typename arrow_result = Value*;
  arrow_result operator->(Ptrlike);
};
```

The following model templates for pointers provide the base cases for Arrowable:

```
template<typename T>
concept Arrowable<const T*, T>
{
  typedef const T* arrow_result;
};

template<typename T>
concept Arrowable<T*, T&>
{
  typedef T* arrow_result;
};
```

```
template<typename T>
concept Arrowable<T*, const T&>
{
  typedef T* arrow_result;
};

template<typename T>
concept Arrowable<const T*, T&>
{
  typedef const T* arrow_result;
};
```

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 requirements common to both Input Iterator and Basic Output Iterator (a concept for the common case of an output iterator whose reference type has a non-polymorphic = operator). The Iterator Associated Types concept demonstrates how associated types can have default definitions:

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

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.

```
template<typename PtrLike, typename Value>
struct concept Dereferenceable
{
  Value operator*(PtrLike&);
};
```

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 [Int98]

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<typename X>
concept InputIterator : IteratorAssociatedTypes<X>,
                        CopyConstructible<X>,
                        Assignable<X>,
                        EqualityComparable<X> {
  where SignedIntegral<difference_type>;
  where Convertible<reference, value_type>;
  where Arrowable<pointer, value_type>;

  typename postincrement_result = X;
  where Dereferenceable<postincrement_result, value_type>;

  pointer operator->(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(const X&);
};
```

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 combination of iterator and value types. Also, note that there are no pointer or difference types for an output iterator, as these are never used and are often not defined to sensible values.

Table 74, Output Iterator requirements [Int98]

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<typename X, typename Value>
concept OutputIterator {
  typename value_type = Value;
  typename reference = X::reference;

  where CopyConstructible<X>;
  where Assignable<X>;

  where value_type == Value;
  where Assignable<reference, value_type>;

  typename postincrement_result = X;
  typename postincrement_ref_result = reference;
  where Dereferenceable<postincrement_result,
                        postincrement_ref_result>;
  where VoidAssignable<postincrement_ref_result, value_type>;
  where Convertible<postincrement_result, const X&>;

  reference operator*(X&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
};

template<typename X>
concept BasicOutputIterator : IteratorAssociatedTypes<X>,
                              CopyConstructible<X>,
                              Assignable<X> {
  where Assignable<reference, value_type>;
  typename postincrement_result = X;
  typename postincrement_ref_result = reference;

  where Dereferenceable<postincrement_result,
                        postincrement_ref_result>;
  where VoidAssignable<postincrement_ref_result, value_type>;
  where Convertible<postincrement_result, const X&>;

  reference operator*(X&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
};

template<BasicOutputIterator !X, typename !Value>
where {Convertible<Value, value_type>}
concept OutputIterator<X, Value> {
  typedef Value value_type;
  typedef BasicOutputIterator<X>::reference reference;
};
```

For Forward Iterator and all iterator concepts refining it, there are two variants of each concept: mutable and non-mutable. 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. A diagram of all of the ConceptGCC iterator concepts is in Figure 2; in the diagram, solid lines represent requirements and refinements within concepts while dotted lines represent models.

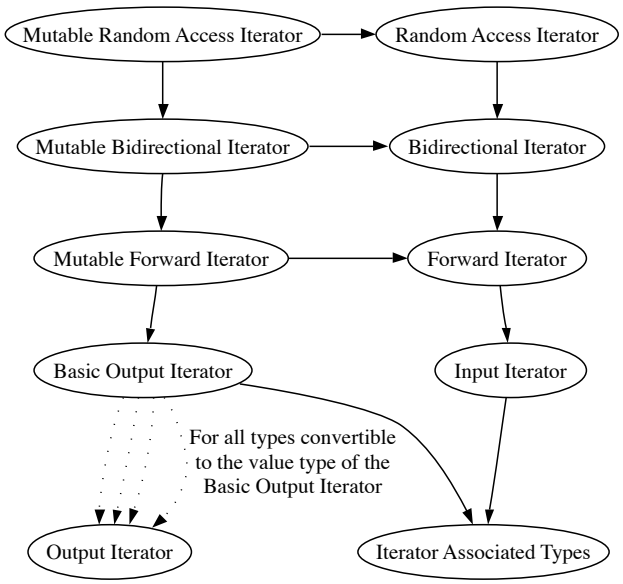

Figure 2: ConceptGCC iterator concepts

Table 75, Forward Iterator requirements [Int98]

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<typename X>
concept ForwardIterator : InputIterator<X>,
                          DefaultConstructible<X> {
  where Convertible<reference, const value_type&>;
  where Arrowable<pointer, const value_type&>;
  where Convertible<postincrement_result, const X&>;
};

template<typename X>
concept MutableForwardIterator : ForwardIterator<X>,
                                 BasicOutputIterator<X> {
  where reference == value_type&;
  where Arrowable<pointer, value_type&>;
};
```

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

Table 76, Bidirectional Iterator requirements [Int98]

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<typename X>
concept BidirectionalIterator : ForwardIterator<X> {
  typename postdecrement_result = X;
  where Dereferenceable<postdecrement_result, value_type>;
  where Convertible<postdecrement_result, const X&>;

  X& operator−−(X&);
  postdecrement_result operator−−(X&, int);
};

template<typename X>
concept MutableBidirectionalIterator :
    BidirectionalIterator<X>,
    MutableForwardIterator<X> {
  where reference == value_type&;
  where Arrowable<pointer, value_type&>;
};
```

Table 77, Random Access Iterator requirements [Int98]

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<typename X>
concept RandomAccessIterator : BidirectionalIterator<X>,
                               LessThanComparable<X> {
  X& operator+=(X&, difference_type);
  X operator+(X, difference_type);
  X operator+(difference_type, X);
  X& operator−=(X&, difference_type);
  X operator−(X, difference_type);
  difference_type operator−(X, X);
  reference operator[](X, difference_type);
};

template<typename X>
concept MutableRandomAccessIterator :
    RandomAccessIterator<X>,
    MutableBidirectionalIterator<X> {
  where reference == value_type&;
  where Arrowable<pointer, value_type&>;
};
```

To improve backward compatibility, a set of model templates are 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&. The IteratorTraits concept provides a simplified way to access iterator_traits. Also, if iterator_traits<Iter> is not valid for some type Iter, there is no error: the model IteratorTraits<Iter> simply doesn't exist, and so the compatibility models for the new iterator concepts for that type are also disabled without error.

```
template<typename Iter>
struct concept IteratorTraits {
  typename iterator_category = iterator_traits<Iter>::iterator_category;
  typename value_type = iterator_traits<Iter>::value_type;
  typename difference_type = iterator_traits<Iter>::difference_type;
  typename pointer = iterator_traits<Iter>::pointer;
  typename reference = iterator_traits<Iter>::reference;
};

template<IteratorTraits ! Iter>
where { Convertible<iterator_category, input_iterator_tag>}
concept InputIterator<Iter> {
  typedef IteratorTraits<Iter>::value_type value_type;
  typedef IteratorTraits<Iter>::difference_type difference_type;
  typedef IteratorTraits<Iter>::pointer pointer;
  typedef IteratorTraits<Iter>::reference reference;
};

template<IteratorTraits ! Iter>
where { Convertible<iterator_category, input_iterator_tag>,
        Convertible<iterator_category, forward_iterator_tag>}
concept ForwardIterator<Iter> {
  typedef IteratorTraits<Iter>::value_type value_type;
  typedef IteratorTraits<Iter>::difference_type difference_type;
```

```
  typedef IteratorTraits<Iter>::pointer pointer;
  typedef IteratorTraits<Iter>::reference reference;
};

template<IteratorTraits ! Iter>
where { Convertible<iterator_category, input_iterator_tag>,
        Convertible<iterator_category, forward_iterator_tag>,
        reference == value_type&}
concept MutableForwardIterator<Iter> {
  typedef IteratorTraits<Iter>::value_type value_type;
  typedef IteratorTraits<Iter>::difference_type difference_type;
  typedef IteratorTraits<Iter>::pointer pointer;
  typedef IteratorTraits<Iter>::reference reference;
};

// Similar model templates for BidirectionalIterator and
// RandomAccessIterator
```

A.3 Container concepts

As the current standard does not have algorithms using container concepts, they were not implemented in the library for ConceptGCC. Hypothetical versions of them are presented here to show that they can be expressed in ConceptGCC, however. The Sequence concept shows that a concept can require polymorphic functions, including member functions and constructors.

Table 68, Sequence requirements [Int98]

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

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

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

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

A.4 Models

The definition of stack, not yet implemented in the concept-enabled standard library, 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<typename T, BackInsertionSequence Seq = std::deque<T> >
where { CopyConstructible<T>, Assignable<T>, 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();
};
```

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. After the following generic functions are defined, the models of these concepts for stack are implicit because the concepts are structural:

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

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

A.5 Algorithms

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

```
template<InputIterator InputIter, typename OutputIter>
where { OutputIterator<OutputIter, value_type> }
OutputIter copy(InputIter first1, InputIter last, OutputIter out) {
  while (first != last) *out++ = *first++;
  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<InputIterator InputIter, typename OutputIter, typename UnOp>
where { Callable1<UnOp, reference>,
        OutputIterator<OutputIter, result_type> }
OutputIter transform(InputIter first, InputIter last, OutputIter out, UnOp f) {
  while (first != last) *out++ = f(*first++);
  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<InputIterator InputIter1, InputIterator InputIter2,
         typename OutputIter, typename BinOp>
where { Callable2<Func, InputIterator<InputIter1>::value_type,
                        InputIterator<InputIter2>::value_type>,
        OutputIterator<OutputIter, result_type> }
```

```
OutputIter transform(InputIter1 first1, InputIter1 last1, InputIter2 first2, OutputIter out, BinOp f) {
  while (first1 != last1) *out++ = f(*first1++, *first2++);
  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<InputIterator InputIter1, InputIterator InputIter2, typename Cmp>
where { InputIterator<InputIter1>::value_type == InputIterator<InputIter2>::value_type
        StrictWeakOrdering<Cmp, InputIterator<InputIter1>::value_type> }
bool includes(InputIter1 first1, InputIter1 last1, InputIter2 first2, InputIter2 last2, Cmp cmp);
```

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

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

template<BidirectionalIterator Iter>
void advance(Iter& i, difference_type n) {
  while (n > 0) {++i; --n;}
  while (n < 0) {--i; ++n;}
}

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

References

[AS02] David Abrahams and Jeremy Siek. The Boost Operators library. www.boost.org, 2002.

[CKP05] Manuel M. T. Chakravarty, Gabrielle Keller, and Simon Peyton Jones. Associated type synonyms. In *Proceedings of the International Conference on Functional Programming (ICFP '05)*, New York, NY, USA, September 2005. ACM Press.

[CKPM05] Manuel M. T. Chakravarty, Gabrielle Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New York, NY, USA, 2005. ACM Press.

[GJL+03] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 115–134, New York, NY, USA, 2003. ACM Press.

[GJL+05] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 2005. submitted.

[GS05a] Douglas Gregor and Jeremy Siek. Explicit model definitions are necessary. Technical Report N1798=05-0058, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2005.

[GS05b] Douglas Gregor and Jeremy Siek. Implementing concepts. Technical Report N1848=05-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.

[GSW+05] Douglas Gregor, Jeremy Siek, Jeremiah Willcock, Jaakko Järvi, Ronald Garcia, and Andrew Lumsdaine. Concepts for C++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, August 2005.

[Int98] International Organization for Standardization. *ISO/IEC 14882:1998: Programming languages — C++*. Geneva, Switzerland, September 1998.

[JS03] J. Järvi and B. Stroustrup. Mechanisms for querying types of expressions: Decltype and auto revisited. Technical Report N1527=03-0110, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2003. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1527.pdf.

[JWHL03] Jaakko Järvi, Jeremiah Willcock, Howard Hinnant, and Andrew Lumsdaine. Function overloading based on arbitrary properties of types. *C/C++ Users Journal*, 21(6):25–32, June 2003.

[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.

[PJM97] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, June 1997.

[SD05] Bjarne Stroustrup and Gabriel Dos Reis. A concept design (rev. 1). Technical Report N1782=05-0042, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, May 2005.

[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.

[SGG+05] Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, January 2005.

[Sie05] Jeremy Siek. *A Language for Generic Programming*. PhD thesis, Indiana University, 2005.

[SL05a] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, June 2005. ACM Press.

[SL05b] Jeremy Siek and Andrew Lumsdaine. Language requirements for large-scale generic libraries. In *GPCE '05: Proceedings of the fourth international conference on Generative Programming and Component Engineering*, September 2005. accepted for publication.

[SLL01] Jeremy Siek, Andrew Lumsdaine, and Lie-Quan Lee. *Boost Graph Library*. Boost, 2001. http://www.boost.org/libs/graph/doc/index.html.

[SLL02] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[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.