Concepts

Douglas Gregor
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
dgregor@cs.indiana.edu

Bjarne Stroustrup
Department of Computer Science
Texas A&M University
College Station, TX 77843
bs@cs.tamu.edu

Document number: N2042=06-0112
Date: June 24, 2006
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

Contents

1 Introduction

Concepts introduce a type system for templates that makes templates easier to use and easier to write. By checking the requirements that templates place on their parameters prior to template instantiation, concepts can eliminate the spectacularly poor error messages produced by today's template libraries, making them easier to use. On the implementation side, concepts replace a grab-bag of template tricks (including SFINAE, tag dispatching, traits, and some forms of template metaprogramming) with a small set of features designed specifically to support the Generic Programming paradigm that underlies many C++ template libraries, thus making it easier for users to develop robust template libraries. At the core of concepts is the idea of separate type checking for templates. Template declarations are augmented with a set of constraints (requirements): if the definition of the template type-checks against these requirements, and a use of a template meets these requirements, then the template should not fail to instantiate.

This proposal represents a merger of many different design ideas from many different people, many of which appear in a series of committee proposals [3, 4, 5, 12, 13, 14, 15, 16]. The vast majority of this proposal has been implemented in the ConceptGCC [2] compiler. Concepts have been used to upgrade much of the Standard Library to provide better error messages and a cleaner, more robust implementation, illustrating their immediate usefulness to C++ users and their ability to provide excellent backward compatibility with C++03.

Concepts have also been used to express more advanced generic libraries, such as the Boost Graph Library.

1.1 Goals

Concepts are intended to improve the C++ template system and make Generic Programming more accessible. Here we lay out our specific goals when designing a concept system for C++0x.

Make it easier to write generic code Implementing generic libraries in C++03 is far too complicated due to the large number of template techniques required. Concepts should make it easier to write generic code by making templates simpler and safer, without increasing the amount of code that users need to write.

Provide performance as good as "plain old templates" C++ templates provide flexibility while retaining unsurpassed performance, making them ideal for implementing generic, high-performance libraries. The introduction of concepts must not introduce any overhead relative to the existing template mechanism.

Support Generic Programming The Generic Programming paradigm, which was behind the development of the Standard Template Library and drives the design of most generic C++ libraries, gives us a solid foundation on which concepts are based. Concepts should fully support Generic Programming, so that C++ can maintain its dominance as the best language for Generic Programming.

Support most current template uses Current template usage involves:

- built-in types and user-defined types
- concrete types and class hierarchies
- operations as members, as free standing functions, and as member templates,
- operations as operators and as named functions, and as function objects
- parameterization on types, integers, operations, and templates
- reliance on overloading and conversion

Concepts must provide general ways of supporting most of the uses currently considered important for style or performance. Some programming styles/techniques, such as template meta-programming, may not be obvious candidates for concept support, but it should be possible to gain some of the benefits of concepts where complete support is possible.

Maintain backward compatibility We need to retain the current meaning of templates, so that existing programs will continue to compile. More interesting, however, is that we need to enable backward compatibility in C++ template libraries as those libraries are converted to use concepts. As an example, the C++ Standard Library, when upgraded with concepts, should be backward-compatible with the C++03 Standard Library.

Have a comprehensible compilation model Most of the users of concepts will not have PhDs and will not read anything with Greek letters. It should be possible for a reasonably bright programmer to correctly imagine what the memory layout of his data structures are, what functions are called, and what code is replicated during instantiation.

"Concepts" is a language feature, not a confederation of features The various concept mechanisms must fit together seamlessly. There must be no "seams" where something can be expressed in several ways (each reflecting a focus on a single part) or where something can't be done because two parts can't quite exchange needed information.

1.2 Status

This document is in early draft status. It contains many typos and errors, and lacks much introductory information that would aid understanding. At present, it is best viewed as a reference manual for readers that already understand concepts. With future versions of this document, we hope to improve the presentation to provide both an introductory view and a language-technical view of concepts.

We hope this document will help readers use concepts, as currently implemented in ConceptGCC, to gain experience with concepts. Most importantly, we also hope for help in our search for simplifications of the concept-related language features and in our search for more elegant ways of expression concepts and their uses. Among the most active such areas are:

1. Can requirements on operations best be defined in terms of signatures (as used in this document) or as "Use patterns" [16] (as currently being implemented to allow direct comparison)?

2. Is it possible to simplify the refinement and nested requirements (possibly unifying the language facilities supporting those ideas)?

3. Is it possible to improve, simplify, or generalize the "shortcut rule" for qualifying concept members with their template argument name (see "alternative" in Section 3.4.2)?

4. Are concepts still expressive enough if `||` constraints cannot be used (see Section 3.4.1)?

5. What is the most natural semantics for unqualified name lookup within constrained templates (see Section 3.4.3 and its "alternative")?

1.3 Related Documents

This document supercedes all previous concepts proposals [5, 12, 13, 14, 15, 16]. In addition to this proposal, which defines the language features required to support concepts, we have also provided a series of proposals that introduce concepts into the C++0x Standard Library. While not complete at this time, these proposals serve three important purposes. First, they provide many real-world examples of the definition and use of concepts, and can be used as a companion to this proposal. While we endeavor to keep examples short and simple within this document, the documents describing a concept-enhanced Standard Library contain actual, working definitions with full details. Second, these proposals illustrate that concepts can model existing, non-trivial uses of templates while providing excellent backward compatibility. Finally, concepts as a language feature will not gain widespread acceptance without a concept-enabled Standard Library. The Standard Library will give users their first exposure to concepts in C++0x, providing the canonical examples from which users will learn how best to utilize these new features. If we (the committee) don't use concepts, who will?

The following companion proposals describe changes to the C++0x Standard Library to introduce complete support for concepts:

- Concepts for the C++0x Standard Library: Approach [7]
- Concepts for the C++0x Standard Library: Introduction [8]
- Concepts for the C++0x Standard Library: Utilities [11]
- Concepts for the C++0x Standard Library: Iterators [9]
- Concepts for the C++0x Standard Library: Algorithms [6]
- Concepts for the C++0x Standard Library: Numerics [10]

2 Using Concepts

Concepts essentially provide a type system for templates, allowing the definition of templates to be type-checked separately from their uses. The core feature of concepts, therefore, is to provide a way to state what behaviors template parameters must have, so that the compiler can check them. For instance, consider the min() function from the Standard Library:

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

When can we use sum() with any type T that has a less-than operator taking two values of type T and returning some value convertible to **bool**. So long as those requirements are met, sum<T>() will instantiate properly. These requirements cannot be expressed in C++, so they are typically expressed in documentation as *concepts*. See, e.g., the SGI Less Than Comparable concept documentation at http://www.sgi.com/tech/stl/LessThanComparable.html. Concepts allow us to express these concept requirements directly in the sum() template:

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

Here we have used concepts to place a requirement on the template type parameter T. Instead of stating that T is an arbitrary type via **typename**, we state that it is a LessThanComparable type, meaning that min() will only accept parameters whose types meet the requirements of the LessThanComparable concept. From the user's point of view, the concept-based min() is almost identical to its predecessor: for any LessThanComparable type, it works in the same way. However, there is a large difference when min() is called with a type that is **not** LessThanComparable: instead of failing to instantiate min() (when no suitable **operator**< can be found), the error is detected at the call to min() when the LessThanComparable requirements cannot be satisfied. This early detection of errors is the reason that concepts improve error messages, since we no longer need to produce an instantiation stack or direct users to code inside a library's implementation.

2.1 Concepts

How does the LessThanComparable concept describe its requirements? In many cases, we need only list the signatures of functions and operators we want to have within the definition of the concept. Here is the definition of LessThanComparable:

```
auto concept LessThanComparable<typename T> {
  bool operator<(T, T);
};
```

When defining a concept, we use the keyword **concept** followed by the name of the concept and a template parameter list. In this case, the LessThanComparable concept only has one parameter, the type for which we want **operator**< defined. Inside the body of the concept, we list the declarations that we expect to have, i.e., an **operator**< that takes two T parameters and returns a **bool**. The **auto** specifier means that *any* type which has a suitable **operator**< will be considered LessThanComparable; if omitted, the user will have to explicitly state that her types meet the requirements of the concept using a *concept map* (see Section 2.2).

Concept *signatures* can describe many different kinds of requirements for functions, constructors, operators, member functions, member function templates, etc. The following concept, Regular, illustrates many of these requirements. The Regular concept applies to well-behaved types that can be constructed, copied, compared, etc.

```
auto concept Regular<typename T> {
  T::T();                          // default constructor
  T::T(const T&);                  // copy constructor
  T::~T();                         // destructor
  T& operator=(T&, const T&);      // copy assignment
  bool operator==(T, T);           // equality comparison
  bool operator!=(T, T);           // inequality comparison
  void swap(T&, T&);               // swap
};
```

2.1.1 Multi-Type Concepts and Where Clauses

Concepts can describe requirements on multiple types simultaneously. For instance, we often need to express that two types aren't necessarily equal, but one can be converted to another. This requirement can be expressed with a two-parameter concept Convertible:

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

To use this concept, we need to employ the general form of describing the constraints on a template, called a *where clause*. Where clauses can contain any number of concept constraints that augment the constraints placed in the template header. For instance, we can define a function template that converts from one type to another:

```
template<typename U, typename T>
  where Convertible<T, U>
  U convert(const T& t) {
    return t;
  }
```

convert() can be used for any valid conversion, e.g., convert<**float**>(17) is valid (**int** is convertible to **float**) but convert<**int**∗>(17.0) is not (a **float** is not convertible to an **int** pointer). We'll come back to **where** clauses later.

With multi-type concepts, we have the tools to express rudimentary iterator concepts, like those in the Standard Library. For instance, we could make Iterator a two-parameter concept, one for the iterator type itself and one for the value type of the iterator:

```
concept InputIterator <typename Iter, typename Value> {
  Iter :: Iter ();                   // default  constructor
  Iter :: Iter (const Iter &);       // copy constructor
  Iter ::˜ Iter ();                  // destructor
  Iter & operator=(const Iter&);     // copy assignment
  Value operator∗(Iter);             // dereference
  Iter & operator++(Iter&);          // pre-increment
  Iter  operator++(Iter&, int);      // post-increment
  bool operator==(Iter, Iter);       // equality  comparison
  bool operator!=(Iter, Iter );      // inequality  comparison
  void swap(Iter , Iter );           // swap
}
```

We can even define simple algorithms with this concept:

```
template<typename Iter, Regular V>
  where InputIterator<Iter, V>
  Iter find(Iter first, Iter last, const V& value) {
    while (first != last && ∗first != value)
      ++first;
    return first;
  }
```

2.1.2 Composing Concepts

Looking at the InputIterator and Regular concepts, there is a lot of overlap. In fact, every requirement in the Regular concept is also a requirement on the Iter type parameter to the InputIterator concept! What we really want to say is that the Iter type of the InputIterator concept is Regular, which we can do with a *nested requirement*. Nested requirements are like **where** clauses, but they go inside the body of the concept as follows:

```
concept InputIterator<typename Iter, typename Value> {
  where Regular<Iter>;
  Value operator*(Iter); // dereference
  Iter& operator++(Iter&); // pre-increment
  Iter operator++(Iter&, int); // post-increment
}
```

Any concept requirements can be composed in this manner. Sometimes, however, there is a more fundamental, *hierarchical* relationship between two concepts. For instance, every RandomAccessIterator is a BidirectionalIterator, every BidirectionalIterator a ForwardIterator, and every ForwardIterator an InputIterator. *Concept refinement* allows us to express these hierarchical relationships using a syntax akin to inheritance:

```
concept ForwardIterator<typename Iter, typename Value>
  : InputIterator<Iter, Value> {
  // no syntactic differences, but adds the multi-pass property
}
```

Concept refinement has one particularly important property: every type that meets the requirements of the refining concept (ForwardIterator) also meets the requirements of the refined concept (InputIterator), so my forward iterators can be used in an algorithm that requires input iterators, such as find(), above.

When composing concepts, should you use nested requirements or refinement? If there is a direct hierarchical relationship, use refinement; otherwise, use nested requirements.

2.1.3 Associated Types

Standard Library aficionados will note that the InputIterator concept provided is overly simplified. In particular, the value type of an iterator (represented by the parameter Value) is only one of four "extra" types for an iterator. To be more precise, we really should declare InputIterator as:

```
concept InputIterator<typename Iter, typename Value, typename Reference,
                      typename Pointer, typename Difference> {
  where Regular<Iter>;
  where Convertible<Reference, Value>;
  Reference operator*(Iter); // dereference
  Iter& operator++(Iter&); // pre-increment
  Iter operator++(Iter&, int); // post-increment
  // ...
}
```

Unfortunately, these extra concept parameters come with a cost: every time we use the InputIterator concept, we need to declare template parameters for every concept parameter. This new, more proper formulation of InputIterator forces a lot of complexity into our previously-simple find() algorithm:

```
template<typename Iter, Regular V, typename R, typename P, typename D>
  where InputIterator<Iter, V, R, P, D>
  Iter find(Iter first, Iter last, const V& value) {
    while (first != last && *first != value)
      ++first;
    return first;
  }
```

Associated types allow us to declare auxiliary types like Reference, Pointer, Difference and even Value inside the concept body, minimizing the number of concept parameters and simplifying algorithms. With associated types, we can make InputIterator a single-parameter concept and keep the definition of find() simple:

```
concept InputIterator<typename Iter> {
  typename value_type;
  typename reference;
  typename pointer;
  typename difference_type;
  where Regular<Iter>;
  where Convertible<reference_type, value_type>;
  reference operator*(Iter); // dereference
  Iter& operator++(Iter&); // pre-increment
  Iter operator++(Iter&, int); // post-increment
  // ...
}

template<InputIterator Iter>
  where Regular<Iter::value_type>
  Iter find(Iter first, Iter last, const Iter::value_type& value) {
    while (first != last && *first != value)
      ++first;
    return first;
  }
```

There is some new syntax in this example. We have moved the InputIterator requirement back into the template header (InputIterator Iter). Then we state that the value_type of Iter, written Iter::value (no **typename** required!) is Regular. Finally, we require that the third parameter to find() be of type Iter::value_type. Note how associated types have helped us simplify the find() template, because we only need to refer to the associated types that we are interested in.

Associated types can be used for many reasons, but much of the time they are used to express the return types of signatures that are otherwise unknown. For instance, the Standard Library's transform() operation accepts a binary function, for which we could write the following concept:

```
auto concept BinaryFunction<typename F, typename T1, typename T2> {
  typename result_type;
  result_type operator()(F&, T1, T2);
};
```

The **operator**() signature in this concept says that a value of type F can be invoked with two arguments of type T1 and T2, respectively. The BinaryFunction concept does not specify what **operator**() must return, so it contains an associated type that gives a name to this return value (result_type) but places no requirements on the associated type. Many uses of BinaryFunction will, of course, introduce their own requirements on the type. For instance, the following declaration of transform() uses result_type as the value_type of an OutputIterator:

```
template<InputIterator InIter1, InputIterator InIter2,
         class OutIter, class BinOp>
  where BinaryFunction<BinOp, InIter1::reference, InIter2::reference> &&
        OutputIterator<OutIter,
                       BinaryFunction<BinOp, InIter1::reference, InIter2::reference>::result_type>
  OutIter transform(InIter1 first1, InIter1 last1,
                    InIter2 first2, OutIter result,
                    BinOp binary_op);
```

By using associated types to describe return types for which we have no requirements, we also open the door for refinements to add requirements on those return types. For instance, a binary predicate is just a binary function whose return type can be interpreted as a boolean value:

```
auto concept BinaryPredicate<typename F, typename T1, typename T2>
  : BinaryFunction<F, T1, T2> {
  where Convertible<result_type, bool>;
};
```

The reader may recognize that we use associated types in the same places where existing C++ libraries use traits: in fact, concepts and associated types replace the need for traits with a simpler, safer mechanism. Associated types can even be accessed with a more general syntax that is reminescent of traits, e.g., InputIterator<Iter>::value_type (again, no **typename**), which is necessary for multi-type concepts.

2.2 Concept Maps

Concepts describe the interfaces required for generic algorithms and data structures to work properly, but sometimes it isn't clear when—or how—certain types meet the interface requirements of a concept. *Concept maps* allow users to specify when their types meet the requirements of a concept (required when the concept is not marked **auto**), but also allow one to adapt the syntax of existing types to the syntax expected by the concept *without changing* *the definition of the types.* For instance, we can make all **char** pointers into InputIterators with a simple concept map:

```
concept_map InputIterator<char*> {
  typedef char               value_type ;
  typedef char&              reference ;
  typedef char*              pointer ;
  typedef std:: ptrdiff_t    difference_type ;
};
```

This concept states that **char**∗s are concept maps (allowing us to call algorithms requiring InputIterators with character pointers) and provides definitions for each of the associated types in the concept. The remaining requirements–default constructibility, copy constructibility, increment, dereference, etc.—will be implicitly defined by the compiler.

If the implicit definitions of certain operations will not work or are incorrect, concept maps may provide their own function definitions instead. This syntax adaptation allows us to view types differently when seen through concepts, so that a type can expose many different interfaces without having those interfaces clash. You can think of concept maps as a pair of colored glasses that changes how you see types without changing the actual type. For instance, we could imagine looking at an integer as an iterator, where the dereference operator is the identity function:

```
concept_map InputIterator<int> {
  typedef int value_type;
  typedef int reference;
  typedef int* pointer;
  typedef int difference_type;

  int operator*(int x) { return x; }
};
```

Looking at **int** through our InputIterator glasses, we see that incrementing the **int** moves to the next value, dereferencing an **int** return itself (so the sequence just produces integer values), and all of the other iterator requirements are met. We can now use integers when calling Standard Library algorithms, e.g., to write a series of values through cout:

```
copy(1, 17, std::ostream_iterator<int>(std::cout, " "));
// prints: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

Like many other constructs in C++, concept maps can also be templated. Since *all* pointers are InputIteratorss (not just character pointers!), we should instead write a concept map template:

```
template<typename T>
concept_map InputIterator<T*> {
  typedef T                  value_type ;
  typedef T&                 reference ;
  typedef T*                 pointer ;
  typedef std:: ptrdiff_t    difference_type ;
};
```

The real power of concept maps comes in when we combine concept map templates with syntax adaptation, composing concept maps so that we can look at data types through a series of lenses piled on top of one another. Consider, for instance, a concept Stack, which only allows one to push values, pop values, query the top value, and determine when the stack is empty:

```
concept Stack<typename X> {
  typename value_type;

  void push(X&, value_type);
  void pop(X&);
  value_type top(const X&);
  bool empty(const X&);
};
```

The Stack concept can be implemented by many different data structures, among them is std::vector. To do so, we need only transform some syntax:

```
template<typename T>
concept_map Stack<std::vector<T> > {
  typedef T value_type;

  void push(std::vector<T>& v, T x)       { v.push_back(x); }
  void pop(std::vector<T>& v)             { v.pop_back(); }
  T top(const std::vector<T>& v)          { return v.back(); }
  bool empty(const std::vector<T>& v)     { return v.empty(); }
};
```

Now, every std::vector is a Stack. What about std::list and std::deque? These other data structures provide the same Stack-like members as std::vector, so we would need to write identical concept maps for them. Shouldn't we be able to write concept maps that support all of these containers?

The containers std::vector, std::list, and std::deque shared much of their interface. In fact, most of their interface is described by the BackInsertionSequence concept, an excerpt from which is shown below:

```
concept BackInsertionSequence<typename X> {
  typename value_type = X::value_type;
  void X::push_back(value_type);
  void X::pop_back();
  value_type& X::back();
  const value_type& X::back() const;
  bool X::empty() const;
};
```

Since we can implement a Stack with any data structure that provides suitable push_back, pop_back, back, and empty functions (like std::vector, std::list, and std::deque), we can implement a Stack with any data structure that meets the requirements of a BackInsertionSequence. The concept map would be written as follows:

```
template<BackInsertionSequence X>
```

```
concept_map Stack<X> {
  typedef X::value_type  value_type ;

  void push(X& x, value_type value)        { x. push_back(value); }
  void pop(X& x)                           { x. pop_back(); }
  T top(const X& x)                        { return x. back(); }
  bool empty(const X& x)                   { return x. empty(); }
};
```

How is an std::list a Stack? Well, we know that std::list is a BackInsertionSequence, because that is defined by the standard. The concept map above says that every type X that is a BackInsertionSequence is also a Stack, with some adaptation to the syntax. In the glasses analogy, we see through the Stack concept which bends the syntax to that of the BackInsertionSequence concept, which we see through to the actual implementation of std::list. By templating concept maps in terms of other concepts, we allow users to "stack" lenses together and allow one to seamlessly bridge from one set of concepts into another.

2.3 Constrained Templates

Templates that use concepts are actually of a new kind of template, called a *constrained template*. Constrained templates are templates that contain a concept constraints on their template parameters. Constrained templates are fully type-checked at the time of definition, so that errors in the template definition will be caught early. For instance, say we tried to write a max() function as follows:

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

This definition of max() is ill-formed because it uses the > operator in its body. Without concepts, this error would only be detected when a user tries to call max() with a type that provides < but not >. With concepts, however, this template definition is ill-formed: when the function body is initially parsed, the compiler will attempt to find any > operator that takes two parameters of type T. The compiler will search inside the LessThanComparable concept (which provides a < operator, only) and the global scope, but there is no such operator. Thus, the definition of the template is ill-formed.

On the other hand, when type-checking the min() template, the search for an **operator**< to satisfy x < y will match the **operator**< in LessThanComparable. And, since that **operator**< returns a **bool** value, the min() will type-check properly. By type-checking both the definitions and uses of a template against the same set of requirements, concepts can eliminate nearly all instantiation-time failures.

2.4 Concept-based Overloading

Some algorithms can be implemented in several different ways depending on the capabilities of their input types. The canonical example of this ability is the advance() algorithm from the

Standard Library, which moves an iterator i forward n steps. For input iterators, advance() can only step the iterator forward n times, requiring a linear number of operations. With bidirectional iterators, advance() can move either forward ($n > 0$) or backward ($n < 0$), but still requires a linear number of operations to do so. Random access iterators are the most powerful iterators, because they can jump any number of steps in a single operation. *Concept-based overloading* allows us to provide multiple versions of the same algorithm, each with different concept requirements, and ensures that the most specific algorithm will be picked when called. Using concept-based overloading is as simple as writing down the different implementations of the algorithm:

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

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

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

Now, if we call advance with a pointer and an offset, the compiler will select the third (most efficient) algorithm, because pointers are RandomAccessIterators. On the other hand, if we call advance with an iterator into an std::list, the compiler will select the second algorithm, because the third algorithm is not supported. The behavior of the advance() function above should look familiar: it does the same thing as advance() in the C++03 Standard Library, but instead of implementing concept-based overloading using template tricks—e.g., tag dispatching or SFINAE—concepts provide the precise feature we need to solve this problem.

Concept-based overloading is actually a form of specialization, affecting the partial ordering rules for function and class templates alike. Thus, one could provide different versions of a class template based on the concept requirements that its parameters meet:

```
template<EqualityComparable T>
class dictionary {
  // slow, linked-list implementation
};

template<LessThanComparable T>
where !Hashable<T>
class dictionary<T> {
  // balanced binary tree implementation
};

template<Hashable T>
class dictionary<T> {
    // hash table implementation
};
```

For this dictionary template , we'll select among three different implementations—a linked list, a balanced binary tree, or a hash table—based on what operations the type T supports. Näively writing down the three operations would leave us with an ambiguity: what if a type T was both LessThanComparable and Hashable? To resolve the ambiguity, we use a *negative constraint*, which states that the second version of dictionary (for balanced binary trees) will only be selected if the type T is not Hashable, thereby preferring the hash table implementation when possible. Thus, concept-based overloading and specialization allows multiple versions of templates to provide the same capabilities in different ways, and negative constraints help the decision-making process by resolving ambiguities.

3 Proposed Language Features

This section describes concepts, their syntax, semantics, and related features. Written in a style part-way between the Annotated C++ Reference Manual [1] and actual proposed text, it is intended as a reference manual for users and implementors alike. Future proposals will refine and clarify this text, with separate documents containing proposed text.

3.1 New Keywords

This proposal introduces five new keywords: **concept**, **concept_map**, **where**, **axiom**, and late_check. All of these keywords will also be reserved words.

3.2 Concepts

declaration:
    concept-definition

concept-definition:
    `auto`$_{opt}$ `concept` *identifier* `<` *template-parameter-list* `>` *refinement-clause*$_{opt}$
        where-clause$_{opt}$ *concept-body* `;`$_{opt}$

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

requirement-specification:
    signature requirement-specification$_{opt}$
    associated-req requirement-specification$_{opt}$
    nested-req requirement-specification$_{opt}$
    axiom-definition requirement-specification$_{opt}$

concept-id:
    template-id

Concepts describe an abstract interface that can be used to constrain templates. Concepts state certain syntactic and semantic requirements on a set of template type, non-type, and template parameters. When used to describe the interface of a constrained template, the requirements have two roles. First, when a constrained template is defined, the requirements of the concept act as declarations, ensuring that the template body is fully type-checked. Second, when a constrained template is used, the template arguments provided to the template must meet the requirements of the concept. By playing these two roles, concepts permit nearly-perfect separate type checking for templates.

1. A *concept-id* is a *template-id* whose *template-name* refers to the name of a concept. [**Example:** CopyConstructible<**int**> is a *concept-id* if CopyConstructible is a concept.]

2. A concept without a preceding **auto** specifier is an *explicit* concept. When a concept map is required for an explicit concept, it must be found through normal concept map lookup (§ 3.3.5).

3. A concept with a preceding **auto** specifier is an *implicit* concept. When a concept map is required for an implicit concept but no concept map can be found through normal concept map lookup (§ 3.3.5), it shall be implicitly generated.

4. Concepts shall only be defined at namespace or global scope.

5. The *template-parameter-list* of a concept shall not contain any requirements specified in-line (§ 3.4.2).

6. The **where** clause of a concept places extra requirements on the concept parameters that must be satisfied by the arguments. It is functionally equivalent to nested requirements in concepts (§3.2.4).

3.2.1 Refinements

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

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

refinement-specifier:
  $::_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

Refinements specify an inheritance relationship among concepts. Concept refinement inherits all requirements, so that the requirements of the refining concept are a superset of the requirements of the refined concept.. Thus, when a concept B refines a concept A, every set of template arguments that meets the requirements of B also meets the requirements of A. [**Example**: In the following example, ForwardIterator refines InputIterator. Thus, any type that meets the requirements of ForwardIterator (say, **int**∗) also meets the requirements of InputIterator.

```
concept InputIterator<typename Iter> { /* ... */ };

concept ForwardIterator<typename Iter> : InputIterator<Iter> { /* ... */ };

concept_map ForwardIterator<int*> { /* ... */ };

template<typename Iter> where InputIterator<Iter> void f(Iter);

f((int*)0); // okay, since concept_map ForwardIterator<int*> implies InputIterator<int*>
```

– **end example**]

1. When either qualified or unqualified lookup into a concept does not find a declaration for a given name, name lookup searches all refined concepts for the name. [**Example**:

   ```
   concept InputIterator<typename Iter> { typename difference_type; };

   concept RandomAccessIterator<typename Iter> : InputIterator<Iter> {
     difference_type operator-(Iter x, Iter y); // okay
   };
   ```

   – **end example**]

2. Names declared by a concept do not hide names declared in the concepts it refines. A name that refers to an associated type within a concept or any of its refined concepts refers to the same associated type in all refined concepts, i.e., there are no ambiguous associated type names. A name that refers to a signature within a concept or any of its refined concepts refers to an overload set consisting of all signatures declared with that name in the concept and all of its refinements. A name shall not refer to both an associated type and a signature. [**Example**:

   ```
   concept A<typename T> {
     typename result_type;
     result_type operator+(T,T);
   };

   concept B<typename U> {
     typename result_type;
     result_type operator+(U);
   };

   concept C<typename V> : A<V>, B<V> {
     // operator+ refers to overload set {A<V>::operator+, B<V>::operator+}

     // result_type refers to the (unique) associated type ‘‘result_type’’, even
     // though it has beed declared in both A and B.
   };

   concept D<typename X> {
     void result_type(X);
   };

   concept E<typename Y> : C<Y>, D<Y> { // error: result_type is ambiguous
     // ...
   };
   ```

– **end example**]

3. The *concept-id*s refered to in the refinement clause shall correspond to fully defined concepts. [**Example**: The following example is ill-formed because C is not fully-defined prior to the refinement clause.

   ```
   concept C<typename T> : C<vector<T>> {/* ... */ }; // error: recursive refinement
   ```

   – **end example**]

4. A *concept-id* in the refinement clause shall not refer to any associated types.

5. A *concept-id* in the refinement clause shall refer to at least one of the template parameters of the concept being defined. [**Example**: The following example is ill-formed because it attempts to refine from a concrete concept map.

   ```
   concept InputIterator<typename Iter> : Incrementable<int> { /* ... */}; // error
   ```

   – **end example**]

3.2.2 Signatures

signature:
  `default`$_{opt}$ *simple-declaration*
  `default`$_{opt}$ *function-definition*
  `default`$_{opt}$ *template-declaration*

Signatures describe functions, member functions, or operators that can be used by constrained templates. Signatures take the form of function declarations, which are present within the definition of a constrained template for the purposes of type-checking. Concept maps for a given concept must provide, either implicitly or explicitly, definitions for each signature in the concept (§ 3.3.1).

1. Signatures can specify requirements for free functions and operators. [**Example**:

   ```
   concept C<typename T> {
     bool operator==(T, T);
     bool operator!=(T, T);
     T invert(T);
   };
   ```

   – **end example**]

2. Signatures shall specify requirements for operators as free functions, even if those operators cannot be overloaded outside of a class. [**Example**:

   ```
   concept Convertible<typename T, typename U> {
     operator U(T);
     V::operator U*() const; // error: cannot specify requirement for member operator
   };
   ```

– **end example**]

3. Signatures can specify requirements for member functions, including constructors and destructors. [**Example**:

```
concept Container<typename X> {
  X::X(int n);
  X::~X();
  bool X::empty() const;
};
```

– **end example**]

4. Signatures can specify requirements for templated functions and member functions. [**Example**:

```
concept Sequence<typename X> {
  typename value_type;
  template<InputIterator Iter>
    where Convertible<InputIterator<Iter>::value_type, Sequence<X>::value_type>
    X::X(Iter first, Iter last);
};
```

– **end example**]

5. Concepts may contain overloaded signatures, but signatures shall have distinct types. [**Example**:

```
concept C<typename X> {
  void f(X);
  void f(X, X); // okay
  void f(X); // error: redeclaration of function 'f'
};
```

– **end example**]

6. All arguments to signatures are passed by reference. Each parameter type T in a signature will be transformed into T **const**&. [**Example**:

```
concept C<typename X> {
  void f(X);
  void f(const X&); // error: redeclaration of function 'f'
};
```

– **end example**]

7. Signatures may have a default implementation. This implementation will be used when implicit generation of an implementation fails (§ 3.3.3). [**Example**:

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
  bool operator!=(T x, T y) { return !(x == y); }
};

class X{};
bool operator==(const X&, const X&);

concept_map EqualityComparable<X> { }; // okay, operator!= uses default
```

– **end example**]

Signatures with a default implementation must be preceded by the `default` keyword.

8. Signatures can be identified via qualified names, but their addresses cannot be taken. [**Example**:

```
concept C<typename T> {
  void twiddle(T);
};

template<C T>
void f(T x) {
  C<T>::twiddle(x); // okay, same as ''twiddle(x)''
  &C<T>::twiddle; // error: cannot take the address of a signature
};
```

– **end example**]

3.2.3 Associated Types, Values, and Template Requirements

associated-req:
    template-parameter ;

Associated types, values, and templates are auxiliary "implicit" parameters used in the description of a concept. They are like template parameters, in the sense that they vary from one use of a concept to another, and nothing is known about them unless constraints are explicitly placed on them via a **where** clause. Associated types are by far the most common, because they are often used to refer to the return types of signatures. [**Example**:

```
concept Callable1<typename F, typename T1> {
  typename result_type;
  result_type operator()(F, T1);
};
```

– **end example**]

1. Associated types, values, and template parameters may be provided with a default value. This default value will be used in a concept map when no corresponding definition is provided (§ 3.3.2). [**Example**:

```
concept Iterator<typename Iter> {
  typename difference_type = int;
};

concept_map Iterator<int*> { }; // okay, difference_type is int
```

– end example]

2. The default value of associated types, values, and templates shall only be type-checked when a concept map implicitly defines the associated type.

3. Associated values are integral constant expressions.

3.2.4 Nested Requirements

nested-req:
    where-clause ;

Nested requirements place additional concepts on the template parameters and associated types, values, and templates of a concept. Nested requirements have the same form and behavior as **where** clauses for constrained templates.

1. Nested requirements can be used to constrain associated types, values, and templates in a concept. [**Example**:

```
concept Iterator<typename Iter> {
  typename difference_type;
  where SignedIntegral<difference_type>;
};
```

– end example]

2. Inline requirements (3.4.2) may be used to define associated types. [**Example**:

```
concept Iterator<typename Iter> {
  SignedIntegral difference_type;
};
```

– end example]

3.2.5 Axioms

axiom-definition:
  `axiom` *identifier* `(` *parameter-declaration-clause* `)` *axiom-body*

axiom-body:
  { *axiom-seq*$_{opt}$ }

axiom-seq:
  axiom axiom-seq$_{opt}$

axiom:
  expression-statement
  `if (` *condition* `)` *expression-statement*

Axioms allow the expression of the semantics required by concepts. Although axioms must be are type-checked at the time of definition, it is unspecified whether axioms have any effect on the semantics of the program. [**Example**:

```
concept Semigroup<typename Op, typename T> {
  T operator()(Op, T, T);

  axiom Associativity(Op op, T x, T y, T z) {
    op(x, op(y, z)) == op(op(x, y), z);
  }
};

concept Monoid<typename Op, typename T> {
  T identity_element(Op);

  axiom Identity(Op, T x) {
    op(x, identity_element(op)) == x;
    op(identity_element(op), x) == x;
  }
};
```

– end example]

1. The equality (==) and inequality (!=) operators are provided for the purposes of type-checking axioms. The following two declarations are considered to be in scope for type-checking axioms:

   ```
   template<typename T> bool operator==(const T&, const T&);
   template<typename T> bool operator!=(const T&, const T&);
   ```

   [**Example**:

   ```
   concept CopyConstructible<typename T> {
     T::T(const T&);
     axiom CopyEquivalence(T x) {
       T(x) == x; // okay, uses implicit == for type-checking
     }
   };
   ```

– **end example**]

2. Where axioms state the equality of two expressions, implementations are permitted to replace one expression with the other. [**Example**:

```
template<typename Op, typename T> where Monoid<Op, T>
  T identity(const Op& op, const T& t) {
    return op(t, identity_element(op)); // can compile as ‘‘return t;’’
  }
```

– **end example**]

3. Axioms can state conditional semantics using **if** statements. When the condition can be proven true, implementations are permitted to apply program transformations based on the *expression-statement*. [**Example**:

```
concept PartialOrder<typename Op, typename T> {
  bool operator()(Op, T, T);

  axiom Reflexivity(Op op, T x) { op(x, x) == true; }
  axiom Antisymmetry(Op op, T x, T y) { if (op(x, y) && op(y, x)) x == y; }
  axiom Transitivity(Op op, T x, T y, T z) { if (op(x, y) && op(y, z)) op(x, z) == true; }
}
```

– **end example**]

3.3 Concept Maps

declaration:
    concept-map-definition

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

concept-map-definition:
    `concept_map` *concept-id* { *concept-map-member-specification*$_{opt}$ } ;$_{opt}$

concept-map-member-specification:
    simple-declaration concept-map-member-specification$_{opt}$
    function-definition concept-map-member-specification$_{opt}$
    template-declaration concept-map-member-specification$_{opt}$

Concept maps describe how a set of template arguments map to the syntax of concept. Whenever a constrained template is used, there must be a concept map corresponding to each *concept-id* requirement in the **where** clause (§ 3.4.1). This concept map may be written explicitly, instantiated from a concept map template, or generated implicitly (for an implicit concept).

[**Example**:

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

concept_map EqualityComparable<student_record> {
  bool operator==(const student record& a, const student record& b)
  { return a.id == b.id;
};

template<EqualityComparable T> void f(T);

f(student_record()); // okay, have concept_map EqualityComparable<student_record>
```

– **end example**]

1. Concept maps shall provide, either implicitly (§ 3.3.3) or explicitly (§ 3.3.1, 3.3.2), definitions for every signature and associated type, value, and template requirement (§ 3.2.2, 3.2.3) in the corresponding concept or its refinements. [**Example**:

   ```
   concept C<typename T> { T f(T); };
   concept D<typename T> : C<T> { };

   concept_map D<int> {
     int f(int); // okay: matches requirement for f in concept C
   };
   ```

   – **end example**]

2. Concept maps shall be defined in the same namespace as their corresponding concept.

3. Concept maps shall not contain extraneous definitions that do not match any requirement in the corresponding concept or its refinements. [**Example**:

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

   concept_map C<int> {
     int f(int); // error: no requirement for function f
   };
   ```

   – **end example**]

4. At the point of definition of a concept map, all nested requirements (§ 3.2.4) of the corresponding concept shall be satisfied. A concept map for which a nested requirement is not satisfied is ill-formed. [**Example**:

   ```
   concept SignedIntegral<typename T> { /* ... */ }

   concept ForwardIterator<typename Iter> {
     typename difference_type;
     where SignedIntegral<difference_type>;
   }
   ```

```
concept_map SignedIntegral<ptrdiff_t> { };

concept_map ForwardIterator<int*> {
  typedef ptrdiff_t difference_type;
}; // okay: there exists a concept_map SignedIntegral<ptrdiff_t>

concept_map ForwardIterator<file_iterator> {
  typedef long difference_type;
}; // error: no concept_map SignedIntegral<long>
```

– **end example**]

5. The definition of a concept map asserts that the axioms (§ 3.2.5) defined in the corresponding concept (and its refinements) hold. It is unspecified whether these axioms have any effect on program translation.

3.3.1 Signature Definitions

Signatures (§ 3.2.2) are satisfied by function definitions within the body of a concept map. These definitions can be used to adapt the syntax of the concept arguments to the syntax of the type. [**Example**:

```
concept Stack<typename S> {
  typename value_type;
  bool empty(S);
  void push(S&, value_type);
  void pop(S&);
  value_type& top(S&);
}

// Make a vector behave like a stack
template<Regular T>
concept_map Stack<std::vector<T> > {
  typedef T value_type;
  bool empty(std::vector<T> vec) { return vec.empty(); }
  void push(std::vector<T>& vec, value_type value) { vec.push_back(value); }
  void pop(std::vector<T>& vec) { vec.pop_back(); }
  value_type& top(std::vector<T>& vec) { return vec.back(); }
}
```

– **end example**]

1. A function declaration in a concept map matches a signature of the same name when the types of the functions are equivalent after substitution of concept arguments. [**Example**:

```
concept C<typename X> {
  void f(X);
};

concept_map C<int> {
  void f(int) { }
};
```

– **end example**]

2. All arguments to functions in a concept map are passed by reference. Each parameter type T in a signature definition will be transformed into T **const**& prior to matching functions to signatures. [**Example**:

```
concept C<typename X> {
  void f(const X&);
  void g(const X&);
  void h(X);
};

concept_map C<int> {
  void f(const int&) { } // okay: exact match
  void g(int) { } // okay: parameter becomes const int&
  void h(int) { } // okay: parameters become const int&
};
```

– **end example**]

3. Functions declared within a concept map may be defined outside the concept map, in a separate translation unit. [**Example**:

```
// c.h
concept C<typename X> {
  void f(X);
};

concept_map C<int> {
  void f(int);
};

// c.cpp
void C<int>::f(int) {
  // ...
}
```

– **end example**]

4. Function templates declared within a concept map match a signature template when the signature template is at least as specialized as the function template. [**Example**:

```
concept C<typename X> {
  typename value_type;

  template<ForwardIterator Iter>
    where Convertible<Iter::value_type, value_type>
    void X::X(Iter first, Iter last);
};
```

```
concept_map C<MyContainer> {
  typedef int value_type;

  template<InputIterator Iter>
    where Convertible<Iter::value_type, int>
    void X::X(Iter first, Iter last) { ... } // okay: signature is more specialized
};
```

– **end example**]

3.3.2 Associated Type, Value, and Template Definitions

Definitions in the concept map provide types, values, and templates for the "implicit" parameters of concepts (§ 3.2.3). These definitions must meet the nested requirements (§ 3.2.4) stated in the body of the concept.

1. Associated type requirements are satisfied by type definitions in the body of a concept map. [**Example**:

   ```
   concept ForwardIterator<typename Iter> {
     typename difference_type;
   }

   concept_map ForwardIterator<int*> {
     typedef ptrdiff_t difference_type;
   };
   ```

   – **end example**]

2. Associated value requirements are satisfied by variable declarations, which must be provided with an initializer, in the body of a concept map. [**Example**:

   ```
   concept Tuple<typename T> {
     int length;
   }

   template<typename T, typename U>
   concept_map ForwardIterator<std::pair<T, U> > {
     int length = 2;
   };
   ```

   – **end example**]

3. Associated template requirements are satisfied by class template declarations in the body of the concept map. [**Example**:

   ```
   concept Allocator<typename Alloc> {
     template<class T> class rebind;
   }

   template<typename T>
   concept_map ForwardIterator<my_allocator<T> > {
   ```

```
    template<class U>
      class rebind {
        typedef my_allocator<U> type;
      };
  };
```

– **end example**]

3.3.3 Implicit Member Definitions

When the requirements of a concept and its refinements are not satisfied by the definitions in the body of a concept map (§ 3.3.1, 3.3.2), default definitions are implicitly defined from the requirements and their default values.

1. Implicitly-defined functions generated for signatures contain a single statement, which is either a return statement containing a use of the corresponding operator or function (for operator requirements with non-**void** return types) or a single expression statement containing a use of the corresponding operator or function (for operator requirements with a **void** return type). Implicitly-defined functions for signatures do not have linkage and cannot have their addresses taken.

2. The implicitly generated definition for a free function requirement contains an unqualified call to a function of the same name. [**Example**:

```
concept C<typename T> {
  void f(T x);
  T g(T x);
}

concept_map C<int> {
  void f(int x) { f(x); } // implicitly generated
  int g(int x) { return g(x); } // implicitly generated
};
```

– **end example**]

3. The implicitly generated definition for an operator requirement contains a use of the corresponding operator.[1] [**Example**:

```
concept C<typename T> {
  bool operator<(T, T);
  T operator-(T);
}

concept_map C<int> {
  bool operator<(int x, int y) { return x < y; }
  T operator-(T x) { return -x; }
};
```

[1]Note that we do not treat operators like free functions so that operators can match built-in operators or operators defined as members.

– end example]

4. If the implicitly generated definition of a signature fails to type-check, and the signature contains a default implementation, the default implementation is used. [**Example**:

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
  bool operator!=(T x, T y) { return !(x == y); }
}

class X {};
bool operator==(X, X);

concept_map EqualityComparable<X> {
  bool operator==(X x1, X x2) { return x1 == x2; } // implicitly generated
  bool operator==(X x1, X x2) { return !(x1 == x2); } // from default
};
```

– end example]

5. Implicitly-defined associated type definitions shall be provided when the associated type requirement contains a default type expression. The implicitly-defined definition substitutes the concept arguments into the default type expression. [**Example**:

```
concept Iterator<typename Iter> {
  typename difference_type = ptrdiff_t;
};

concept_map Iterator<int*> {
  typedef ptrdiff_t difference_type; // implicitly generated
};
```

– end example]

6. Implicitly-defined associated type definitions shall be provided when an associated type without a default type expression is used as the return value of a signature. The associated type's defined value shall be the return type of the corresponding function in the concept map, which may be implicitly or explicitly defined. [**Example**:

```
concept UnaryFunction<typename F, typename T> {
  typename result_type;
  result_type operator()(F&, T);
};

stuct identity {};
concept_map UnaryFunction<identity, int> {
  int operator()(identity&, int x) { return x; }
  // implicitly generated: typedef int result_type;
};

concept_map UnaryFunction<float (*)(float), float> {
  // implicitly generated: float operator()(float (*&f)(float), float x);
  // implicitly generated: typedef float result_type;
};

template<CopyConstructible T>
concept_map UnaryFunction<T (*)(T), T> {
  // implicitly generated: T operator()(T (*&f)(T), T x);
  // implicitly generated: typedef T result_type;
};
```

– **end example**]

7. Implicitly-defined associated value and template definitions shall be provided when the associated value or template requirement contains a default value. The implicitly-defined definition substitutes the concept arguments into the default value.

3.3.4 Refinements and Concept Maps

When a concept map is defined for a concept C that has a refinement clause, concept maps for each of the refinements of C are implicitly defined from the definition of the concept map for C.[2] [**Example**:

```
concept ForwardIterator<typename Iter> { /* ... */ }
concept BidirectionalIterator<typename Iter> : ForwardIterator<Iter> { /* ... */ }

concept_map BidirectionalIterator<list_iter> { /* ... */ }
// implicitly generates concept_map ForwardIterator<list_iter>
```

– **end example**]

1. Concept map templates will be implicitly defined for refinements only when all of the template parameters of the original concept map are deducible from the refinement. [**Example**:

```
concept Ring<typename AddOp, typename MulOp, typename T>
  : Group<AddOp, T>, Monoid<MulOp, T> { /* ... */ };

template<Integral T>
concept_map Ring<std::plus<T>, std::multiplies<T>, T> { }
// okay, implicitly generates:
template<Integral T> concept_map Group<std::plus<T>, T> { }
template<Integral T> concept_map Monoid<std::multiplies<T>, T> { }

template<Integral T, Integral U, Integral V>
  where MutuallyConvertible<T, U> && MutuallyConvertible<T, V> &&
        MutuallyConvertible<U, V>
  concept_map Ring<std::plus<T>, std::multiplies<U>, V> { }
// above concept_map is ill-formed, cannot implicitly generate:
template<Integral T, Integral U, Integral V>
  where MutuallyConvertible<T, U> && MutuallyConvertible<T, V> &&
        MutuallyConvertible<U, V>
```

[2] It is not necessary that these concept maps be implicitly generated, so long as they can be found and synthesized during concept map lookup.

```
    concept_map Group<std::plus<T>, V> { }
  template<Integral T, Integral U, Integral V>
    where MutuallyConvertible<T, U> && MutuallyConvertible<T, V> &&
           MutuallyConvertible<U, V>
    concept_map Monoid<std::multiplies<U>, V> { }
```

– **end example**]

For refinements that are not implicitly defined:

- Concept maps matching the refined concept map shall have been previously defined. [**Example**:

  ```
  concept Ring<typename AddOp, typename MulOp, typename T>
    : Group<AddOp, T>, Monoid<MulOp, T> { /* ... */ };

  template<Integral T, Integral V> where MutuallyConvertible<T, V>
    concept_map Group<std::plus<T>, V> { }
  template<Integral U, Integral V> where MutuallyConvertible<U, V>
    concept_map Monoid<std::multiplies<U>, V> { }

  template<Integral T, Integral U, Integral V>
    where MutuallyConvertible<T, U> && MutuallyConvertible<T, V> &&
           MutuallyConvertible<U, V>
    concept_map Ring<std::plus<T>, std::multiplies<U>, V> { } // okay
  ```

  – **end example**]
- Concept maps for which implicit concept maps cannot be generated from refinements shall not define functions (§ 3.3.1) for signatures in those refined concepts.

2. If a concept map has been defined explicitly, it will not be defined implicitly due to refinement. [**Example**:

   ```
   concept ForwardIterator<typename Iter> { /* ... */ }
   concept BidirectionalIterator<typename Iter> : ForwardIterator<Iter> { /* ... */ }

   concept_map ForwardIterator<list_iter> { /* ... */ }
   concept_map BidirectionalIterator<list_iter> { /* ... */ }
   // does not implicitly generate concept_map ForwardIterator<list_iter>
   ```

   – **end example**]

3.3.5 Concept Map Lookup

When a concept map is required for a given *concept-id*, concept map lookup selects the most specialized concept map. Concept map lookup is required when using a constrained template (to satisfy a *concept-id* requirement), when verifying that a concept map meets the nested requirements of its corresponding concept, or when performing qualified name lookup into a concept map.

1. Concept map lookup determines which concept map templates match the required *concept-id* by matching the template arguments to the concept with the template arguments to the concept map templates. Concept map lookup selects the most specialized concept map using the same partial ordering rules as for class template partial specializations, extended by partial ordering with **where** clauses (§ 3.4.6). If partial ordering of concept maps results in an error or ambiguity, the program is ill-formed.

2. If no matching concept map is found and the corresponding concept is an explicit concept, concept map lookup fails.

3. If no matching concept map is found and the corresponding concept is an implicit concept, the compiler tentatively generates an empty concept map definition for the *concept-id*. The concept map will be type-checked with all of its definitions implicitly generated (§ 3.3.3). If type-checking succeeds, the concept map definition is generated and concept map lookup succeeds. If type-checking fails, the concept map definition is removed along with any other failed instantiations and concept map lookup fails.

4. A well-formed program may include failures in concept map lookup.

3.4 Constrained Templates

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

member-declaration:
    where-clause member-declaration

Constrained templates are templates that have constraints placed on their template parameters. Unlike existing (unconstrained) templates, constrained templates are completely type-checked at the time of their definition. A constrained template verifies that its template arguments meet its stated constraints prior to instantiation. Thus, a well-formed constrained template will only fail to instantiate under very rare circumstances, providing nearly-perflect separate type checking.

1. A template is *constrained* if any requirements have been placed on its template parameters, either through the presence of a **where** clause (§ 3.4.1) or inline requirements (§ 3.4.2).

2. A template that is not *constrained* is *unconstrained*.

3.4.1 Where Clauses

where-clause:
    `where` *constraint-expr*

constraint-expr:
    constraint-expr `||` *and-constraint*
    and-constraint

and-constraint:
    and-constraint `&&` *not-constraint*
    not-constraint

not-constraint:
    `!` *constraint*
    constraint

constraint:
    `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
    `(` *constraint-expr* `)`

where clauses place constraints (requirements) on the parameters of a template. The constraints described in **where** clauses have two roles. First, they restrict the use of the constrained template to template arguments that satisfy the requirements of the **where** clause. Second, they provide declarations available within the definition of the template that will be used for type-checking.

1. A *concept-id* constraint in a **where** clause requires the existence of a concept map. When type-checking a constrained template, the *concept-id* constraint provides declarations from its associated concept for type-checking the definition of the constrained template. A *concept-id* constraint is satisfied when concept map lookup (§3.3.5) finds a unique concept map. [**Example**:

```
concept Addable<typename X> {
  X operator+(X, X);
}

concept_map Addable<int> { };

template<typename T> where Addable<T> T plus(T x, T y) {
  return x + y; // okay: Addable<T> provides T operator+(T, T);
}

int x = 17, y = 42;
plus(x, y); // okay: Addable<int> requirement is satisfied by concept_map Addable<int>
```

   – end example]

2. A *not* constraint !C<T1, T2, ..., TN> requires that no concept map exist for the specified concept, i.e., concept map lookup (§ 3.3.5) must fail for the given *concept-id*. When type-checking a constrained template, *not* constraints do not introduce any

declarations into the template definition. Their only impact on type-checking is the propagation of the fact that no concept map exists for that concept. [**Example**:

```
concept C<typename T> { /* ... */ }

template<Regular T> where !C<T> void f(T x) {
  f(x); // okay, no concept map for C<T>
}

concept_map C<int> { /* ... */ };
float x;
int y;
f(x); // okay, no concept map C<float>
f(y); // error: there exists a concept map C<int>
```

– **end example**]

3. An *and* constraint C1 && C2 requires that both of the constraints C1 and C2 be satisfied. When type-checking a constrained template containing an *and* constraint, the union of C1 and C2 is available to the template definition. [**Example**:

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
}

concept LessThanComparable<typename T> {
  bool operator<(T, T);
}
template<Regular T> where EqualityComparable<T> && LessThanComparable<T>
  bool leq(T x, T y) {
    return x < y || x == y;
  }
```

– **end example**]

4. An *or* constraint C1 || C2 requires that either the constraint C1 must be satisfied or the constraint C2 must be satisfied, but not both. When type-checking a constrained template containing an *or* constraint, the intersection of C1 and C2 is available to the template definition. [**Example**: The following function cos() accepts either integral or floating-point types.

```
concept Integral<typename T> { /* ... */ }
concept Floating<typename T> { /* ... */ }

template<Regular T> where Integral<T> || Floating<T> T cos(T);

class anomaly { /* ... */ };
concept_map Integral<anomaly> { /* ... */ };
concept_map Floating<anomaly> { /* ... */ };
cos(anomaly()); // error: have both Integral<anomaly> and Floating<anomaly>
```

– **end example**]

5. The canonical representation for compound **where** clauses is disjunctive normal form. Two where clauses are considered equivalent if their disjunctive normal forms are equivalent. [**Example**:

   ```
   concept Integral<typename T> { /* ... */ }
   concept Floating<typename T> { /* ... */ }

   template<typename T> where Regular<T> && (Integral<T> || Floating<T>) T cos(T);

   template<typename T>
     where (Regular<T> && Integral<T> || Regular<T> && Floating<T>)
     T cos(T x) { /* defines function above. */ }

   template<typename T> where Regular && !(!Integral<T> && !Regular<T>)
     T cos(T x) { /* error: redefinition of cos */ }
   ```

   – **end example**]

6. A constrained template with an *or* constraint is equivalent to a set of specialized constrained templates, each of which contains one term from the disjunctive normal form of the **where** clause. [**Example**:

   ```
   concept Integral<typename T> { /* ... */ }
   concept Floating<typename T> { /* ... */ }

   template<Regular T> where Integral<T> || Floating<T> T cos(T) { /* body text */ }
   // equivalent to...
   template<Regular T> where Integral<T> T cos(T) { /* same body text */ }
   template<Regular T> where Floating<T> T cos(T) { /* same body text */ }
   ```

   – **end example**]

7. When a constrained template contains an *or* constraint C1 || C2, the meaning of identifiers shall not differ between C1 and C2.[3] [**Example**:

   ```
   concept C1<typename T> {
     typename inverse;
     where Convertible<T, inverse>;
   }

   concept C2<typename T> {
     T inverse(T);
   }

   template<typename T>
     where C1<T> || C2<T> // error: inverse has different meanings in C1 and C2
     f(T t) {
       inverse(t);
     }
   ```

   – **end example**]

[3]This restriction is intended to ensure that an implementation need only parse a constrained template once, rather than producing several different abstract syntax trees corresponding to each term in the disjunctive normal form.

3.4.2 Inline Requirements

concept-name:
  identifier

type-parameter:
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name identifier*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-name identifier* = *type-id*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id identifier*
  `::`$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id identifier* = *type-id*

Inline requirements offer an alternative way of specifying the constraints on template parameters. Inline requirements support the "type of a type" view of concepts, and are therefore limited to concepts whose first parameter is a template type parameter.

1. With the exception of nested name lookup, an inline requirement C<T1, T2, ..., TN> T or C T stated in a template header is equivalent to a constrained template that declares T as **typename** T in its template header and introduces the *concept-id* constraint C<T, T1, T2, ..., TN> into the **where** clause. [**Example**:

```
concept C<typename T> { /* ... */ }

template<C T> void f(T);
// is equivalent to the following, modulo nested type lookup
template<typename T> where C<T> void f(T);
```

   – **end example**]

2. Given an inline requirement C T or C<T1, T2, ..., TN> T for a constrained template, the members of concept C and its refinements are available through qualified lookup into T.[4] [**Example**:

```
concept InputIterator<typename T> { typename difference_type; /* ... */ }

template<InputIterator Iter>
  void advance(Iter& x, Iter::difference_type n);
  // Iter::difference_type is the same as InputIterator<Iter>::difference_type
```

   – **end example**]

3. When inline requirements are used with multi-type parameters, associated types are only injected into the template type parameter being declared. [**Example**:

```
concept UnaryFunction<typename F, typename T> {
  typename result_type;
  result_type operator()(F&, T);
};

template<typename T, UnaryFunction<T> F>
  F::result_type apply(F& f, const T& t) { return f(t); }
  // F::result_type is the same as UnaryFunction<F, T>::result_type
```

[4]Note that this syntactic shortcut does *not* mandate that T actually have these members nested.

– **end example**]

4. Multi-type parameters may be used with the form C T if all the concept parameters of C beyond the first have default values. [**Example**:

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

   template<EqualityComparable T>
     bool equal(const T& x, const T& y) { return x == y; } // okay
   ```

   – **end example**]

[**Alternative**: Instead of making concept C's members available in the scope of T when given the inline requirement C T, we could make the members of every *concept-id* C<T1, T2, ..., TN> that has T in its argument list (e.g., C<T>, C<U> T, C<T, U>, but not C<U, V>) available in T:

```
concept InputIterator<typename Iter> {
  typename value_type;
}

template<typename Iter>
  where InputIterator<Iter> && CopyConstructible<Iter::value_type>
  Iter::value_type deref(Iter f);
```

In this case, given Iter::value_type the compiler will search for value_type in InputIterator<Iter> (since Iter is an argument) but not CopyConstructible<Iter::value_type>. Thus, it will find InputIterator<Iter>::value_type. In contrast, the current qualified name lookup rules for constrained template parameters will make the above example ill-formed. Both forms work equally well when Iter is declared using an inline requirement:

```
template<InputIterator Iter>
  where CopyConstructible<Iter::value_type>
  Iter::value_type deref(Iter f);
```

This alternative has the advantage that inline requirements are equivalent to where clauses, both for name lookup and semantically. It also treats all parameters to concepts equally, because one can find the members of a *concept-id* C<T1, T2, ..., TN> inside any of the parameters T1, T2, ..., TN.

On the other hand, by making the names of members from several concepts available to all of their template arguments, we increase the risk of an ambiguity:

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

concept BinaryOperation<typename BinOp, typeaname T, typename U> {
  typename result_type;
  result_type operator()(BinOp&, T, U);
```

```
}
template<typename T, typename U, BinaryOperation<T, U> BinOp>
  where Assignable<BinOp>
  BinOp::result_type f(BinOp op, T, U); // error: BinOp::result_type
```

In this code, BinOp::result_type is ambiguous when using the alternative qualified name lookup semantics because the compiler searches in both the inline requirement BinaryOperation<T, U> BinOp and the requirement Assignable<BinOp>, both of which have a result_type. With the current qualified name lookup semantics, we only search inside the inline requirement. Note also that, because all arguments are treated equally, T::result_type will find BinaryOperation<BinOp, T, U>::result_type. – **end alternative**]

3.4.3 Type-Checking Constrained Templates

Constrained templates provide complete type-checking at the time of definition, which we refer to as implementation-side type checking. Combined with client-side type-checking against the requirements in the **where** clause, constrained templates provide "nearly" separate type checking: if a given set of template arguments meets the requirements in the **where** clause of a given constrained template, the corresponding instantiation is guaranteed to succeed unless there exist inconsistent specializations or partial ordering ambiguities. [**Example**:

```
template<InputIterator Iter, typename F> where Callable1<F, reference>
F for_each(Iter first, Iter last, F f) {
  while (first < last) { // error: no '<' operator defined
    f(*first); // okay: Callable1<F, reference>::operator()
    ++first; // okay: InputIterator<Iter>::operator++
  }
  return f;
}
```

– **end example**]

1. Type-checking of a constrained template occurs "as if" each template type parameter, *template-id* whose *template-name* is a template template parameter, and associated type has been replaced by an *archetype*. [5] Archetypes are unique class types that provide only the member functions that occur as requirements in the constrained template (§ 3.4.1, 3.4.2). None of the implicitly defined class members (default constructor, copy constructor, destructor, assignment operator) are provided for archetypes.

2. Types made equivalent by same-type constraints (§ 3.5.1) share the same archetype. [**Example**:

   ```
   concept CopyConstructible<typename T> {
     T::T(const T&);
     T::~T();
   };
   ```

[5] With all dependent types in a constrained template being replaced with non-dependent types, constrained templates type-check like non-templates.

```
template<CopyConstructible T, typename U> where SameType<T, U>
  void f(const T& t)
{
  T copy(t); // okay: T::T(const T&) from CopyConstructible
  U other_copy(t); // okay: U has the same archetype as T
};
```

– **end example**]

3. Dependent names in the body of a constrained template are looked up in the concepts associated with each *concept-id* requirement (including refinements of the associated concept and nested requirements) stated in the **where** clause and as nested requirements. For the determination of dependent names, an expression is type-dependent if it would have been type-dependent in an unconstrained template. [**Example**:

```
concept SignedIntegral<typename T> {
  T::T(int);
  T& operator++(T&);
};

concept EqualityComparable<typename T> {
  bool operator==(T, T);
  bool operator!=(T, T);
};

concept InputIterator<typename Iter> : EqualityComparable<Iter> {
  typename difference_type;
  where SignedIntegral<difference_type>;
  typename value_type;
  Iter& operator++(Iter&);
  value_type operator*(Iter);
};

template<InputIterator Iter>
  where EqualityComparable<Iter::value_type>
Iter::difference_type count(Iter first, Iter last, const Iter::value_type& value) {
  Iter::difference_type result = 0;
  while (first != last) { // okay, EqualityComparable<Iter>::operator!=
    if (*first == value) // okay, EqualityComparable<value_type>::operator==
      ++result; // okay, SignedIntegral<difference_type>::operator++
    ++first; // okay, InputIterator<Iter>::operator++
  }
  return result; // okay, CopyConstructible<difference_type> constructor[6]
}
```

– **end example**]

4. Dependent names not found within the concepts will be looked up in the lexical scope. [**Example**:

[6]This requirement is automatically generated via constraint propagation.

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

template<MutableForwardIterator InIter,
         OutputIterator<InIter::value_type> OutIter>
OutIter rotate_copy(InIter first, InIter middle, InIter last, OutIter result) {
  return copy(first, middle, copy(middle, last, result));
  // okay: ''copy'' refers to #1
}
```

– end example]

5. In a constrained template, overload resolution and the selection of the most-specialized template for a given set of template arguments are both performed based on the requirements of the constrained template. [**Example**:

```
concept InputIterator<typename Iter> { };
concept BidirectionalIterator<typename Iter> : InputIterator<Iter> { };
concept RandomAccessIterator<typename Iter> : BidirectionalIterator<Iter> { };

template<InputIterator Iter> void advance(Iter&, Iter::difference_type); // #1
template<BidirectionalIterator Iter> void advance(Iter&, Iter::difference_type); // #2
template<RandomAccessIterator Iter> void advance(Iter&, Iter::difference_type); // #3

template<BidirectionalIterator Iter>
void f(Iter x) {
  advance(x, 1); // type-checks using #2
}
```

– end example]

[**Alternative**: Name lookup within a constrained template only searches the **where** clause for dependent names. This rule is intended to match the semantics of existing templates closely, because non-dependent names retain their current lookup semantics (ignoring the **where** clause). For dependent names, name lookup searches the **where** clause and, if the name is found, this name hides other declarations with the same name from outer scopes.

There are alternative name lookup rules we could imply. The most interesting of these treats name lookup into the **where** clause as a set of using directives that pull the signature names from the concepts into the scope of the constrained template. This approach differs from the dependent-name approach in two ways. First, it eliminates the distinction between dependent and non-dependent names entirely, so that all unqualified name lookups consider names in the **where** clause. Second, names in the **where** clause will overload names found in the lexical scope, rather than hiding them. **– end alternative**]

3.4.4 Instantiating Constrained Templates

Constrained template instantiation is very similar to instantiation of unconstrained templates. During constrained template instantiation, template parameters are replaced by their corresponding template arguments throughout the template, therefore producing a concrete implementation (function, class, or concept map). However, constrained template instantiation limits the amount of name lookup that occurs relative to unconstrained template instantiation, e.g., argument-dependent lookup is not used when instantiating constrained templates.

1. Instantiation of constrained templates replaces references to signatures and associated types inside a concept with the actual function and type definitions provided by the concept map. The effect is "as if" the constrained template were written with only qualified uses of names within concepts:[7] [**Example**:

   ```
   template<InputIterator Iter, typename F> where Callable1<F, reference>
   F for_each(Iter first, Iter last, F f) {
     while (InputIterator<Iter>::operator!=(first, last)) {
       typedef typename InputIterator<Iter>::reference reference; // exposition only
       Callable1<F, reference>::operator()(f, InputIterator<Iter>::operator*(first));
       InputIterator<Iter>::operator++(first);
     }
     return f;
   }
   ```

   – end example]

2. Instantiation of function calls for dependent names not matched in the **where** clause undergo a second stage of partial ordering to select the most specialized function from a set of candidate functions. Given the "seed" function that was used for type-checking the constrained template (§ 3.4.3), the set of candidate functions includes the seed and all functions in the same scope as the seed that have the same function signature (template parameters, function parameters, return type) as the seed and are more specialized than the seed. [**Example**:

   ```
   template<InputIterator Iter> void advance(Iter& i, Iter::difference_type n); // A
   template<BidirectionalIterator Iter> void advance(Iter& i, Iter::difference_type n); // B
   template<RandomAccessIterator Iter> void advance(Iter& i, Iter::difference_type n); // C

   template<BidirectionalIterator Iter> void foo(Iter i) {
     advance(i, 1); // type-checks against B; candidate set is {B, C}
   }
   ```

   – end example]

3.4.5 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 requirements of the template through the process of *constraint propagation*.

[7] This translation ensures that the syntax adaptation provided by concepts maps is employed during instantiation.

1. For every type T that appears as an argument or return type in a function declarator, the requirement std::CopyConstructible<T> is implicitly generated. [**Example**:

   ```
   template<EqualityComparable T>
   bool eq(T x, T y); // implicitly generates requirement CopyConstructible<T>
   ```

   – **end example**]

2. For every *template-id* X<A1, A2, ..., AN> that appears in the declaration of a constrained template T, where X is also a constrained template, the requirements of X are implicitly generated in T. [**Example**:

   ```
   template<LessThanComparable T> class set { /* ... */ };

   template<CopyConstructible T>
   void maybe_add_to_set(std::set<T>& s, const T& value);
   // use of std::set<T> implicitly generates requirement LessThanComparable<T>
   ```

   – **end example**]

3.4.6 Partial Ordering by Where Clauses

Function, class, and concept map 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. [**Example**:

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

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<typename T> where A<T> void g(T x) { std::cout << "3"; }
template<typename T> where A<T> && C<T> void g(T x) { std::cout << "4"; }

concept_map B<int> { };
concept_map C<int> { };
int main() {
  f(17); // outputs ‘‘2’’
  g(42); // outputs ‘‘4’’
}
```

– **end example**]

1. If a constrained template and an unconstrained are identical modulo the **where** clause and template parameter names, the constrained template is more specialized.

2. If two constrained templates are identical modulo the **where** clause and template parameter names, the templates shall be ordered based on the requirements in the **where** clauses using the following algorithm. Let T_1 and T_2 be the two constrained templates.

   (a) Introduce the requirements from the **where** clause of T_1 into a new environment.

(b) 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.

(c) Repeat the process with a new environment, to determine if T_2 is at least as specialized as T_1.

(d) 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.

3.4.7 Late Checking

unary-expression:
    `late_check` *unary-expression*

elaborated-type-specifier:
    `late_check` *elaborated-type-specifier*

concept-map-definition:
    `concept_map` *concept-id* `late_check` { *concept-map-member-specification*$_{opt}$ } `;`$_{opt}$

block-declaration:
    where-clause `;`

Concepts provide the ability to separately type-check constrained template definitions from their uses. Concepts are expressive enough to express many—but not all—uses of C++ templates. Thus, some template code that relies on certain template tricks will not be expressible inside constrained templates.

The late_check keyword provides an "escape hatch" for constrained templates. A late-checked expression, type, or concept map, marked with the late_check qualifier, is parsed "as if" that expression, type, or concept map was in an unconstrained. Within the late-checked expression, type, or concept map, all template type parameters are treated as dependent types and no type-checking is performed.

1. The presence of a **where** clause within a block introduces requirements into the block that will not be verified until instantiation time.

2. Late-checked expressions are not type-checked until instantiation time. [**Example**:

```
template<CopyConstructible T>
T unsafe_add(T x, T y) {
  auto result = late_check(x+y); // okay, type of result varies
  where Convertible<decltype(result), T>; // assume result convertible to T
  return result; // okay, Convertible<decltype(result), T>::operator T
}
```

   – **end example**]

3. Late-checked types are not type-checked until instantiation time. [**Example**:

```
template<CopyConstructible T>
  late_check typename metafunc<T>::result result_type;
  tricky_meta(T x) {
    typedef late_check typename metafunc<T>::result result_type;
    where Convertible<T, result_type>;
    return x;
  }
```

– **end example**]

4. Late-checked concept maps are not type-checked or verified for consistency with their associated concept until instantiation time. [**Example**:

```
template<IteratorTraits Iter>
  where Convertible<Iter::iterator_category, forward_iterator_tag>
  concept_map ForwardIterator<Iter> { ... };
```

– **end example**]

Note that this "escape hatch", using late-checked expressions, types, and concept maps, is in flux. We are still attempting to determine what the right level of granularity for such a feature is (e.g., is expression-level too fine-grained?) and are gathering interesting examples.

3.5 Header ⟨concepts⟩

The ⟨concepts⟩ header provides "core" concepts that can effect the compilation and type-checking process. All concepts reside in namespace std. Note that this description of the <concepts> header is only a summary; the complete description will be available in a separate document describing changes to the Standard Library [11].

3.5.1 Concept SameType

```
concept SameType<typename T, typename U> { /* unspecified */ };
template<typename T> concept_map SameType<T, T> { /* unspecified */ };
```

The SameType concept states that its two type parameters must refer to precisely the same type.

1. When a constrained template contains a constraint SameType<T, U>, type-checking the use of the template is "as-if" SameType was specified as an explicit concept with only a single concept map:

```
template<typename T> concept_map SameType<T, T> { };
```

[**Example**:

```
template<typename T, typename U> where SameType<T, U> void f(T, U);

int x; long y;
f(x, y); // error: SameType<int, long> does not hold
```

– **end example**]

2. When a constrained template is defined and type-checked, the presence of a same-type constraint makes the two type arguments identical.[8] [**Example**:

```
concept EqualityComparable<typename T> {
  bool operator==(T, T);
}

template<EqualityComparable T, typename U> where SameType<T, U>
  bool f(T t, U u) {
    return t == u; // okay, T and U have the same type.
  }
```

– **end example**]

3. A program that contains a concept map SameType<T, U>, where T $\neq$ U, is ill-formed.

3.5.2 Concept **True**

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

The True concept allows the use of integral constant expressions in **where** clauses.[9] [**Example**:

```
template<Regular T> where True<sizeof(T) <= 128> void f(T);
```

– **end example**]

3.5.3 Concept **CopyConstructible**

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

The CopyConstructible concept applies to types that can be passed as arguments to or returned from a function. Although the definition of the CopyConstructible concept can be expressed entirely in a library, CopyConstructible is a core concept because CopyConstructible constraints are generated from function signatures via constraint propagation (§3.4.5).

[8]The type-checking behavior of the SameType concept is central to the notion of type equality in a compiler and cannot be emulated in a library.

[9]Although the True concept can be defined entirely within a library, it is provided as a core concept to allow implementations to provide improved diagnostics.

3.6 Miscellaneous

This section describes the effects that concepts will have on other parts of the C++ language, motivating each change with examples.

3.6.1 Implicit Declaration of Class Members

When certain special class members are not declared explicitly within a class, these class members—default constructors, destructors, copy constructors, and assignment operators—are implicitly declared and, when necessary, defined. These implicit declarations open up a loophole in the concept system, because the declarations will be implicitly generated even if the definitions will not compile properly. For instance, if a class type X has a private default constructor, DefaultConstructible<X> will not hold; DefaultConstructible<std::pair<X, X> >, however, *will* hold, because the pair default constructor is implicitly declarated, even though its definition will fail to compile.

Users can avoid these problems using **where** clauses on explicitly-defined constructors, destructors, and assignment operators. For instance:

```
template<typename T, typename U>
struct pair {
  where DefaultConstructible<T> && DefaultConstructible<U>
    pair() : first(), second() { }

  where CopyConstructible<T> && CopyConstructible<U>
    pair(const T& t, const U& u) : first(t), second(u) { }

  where Destructible<T> && Destructible<U>
    ~pair() { }

  where Assignable<T> && Assignable<U>
    pair& operator=(const pair<T, U>& other) {
      first = other.first;
      second = other.second;
    }

  T first;
  U second;
};
```

Unfortunately, this change would require modifications to much existing code. Instead, we opt to only declare implicit default constructors (12.1p5), destructors (12.4p3), copy constructors (12.8p4), and copy assignment operators (12.8p10) when the definitions will type-check properly, eliminating the problem without requiring changes to existing code.

3.6.2 Default Arguments

Default function and template arguments will only be type-checked when they are used. For instance, consider the following constructor for the vector class template:

```
template<CopyConstructible T>
class vector {
```

```
public:
  vector(size_t n, const T& value = T());
};
```

If the default argument were type-checked at definition time, the code would be ill-formed: T is not required to have a default constructor. However, this constructor can be used with types that don't have default constructors. One could split this constructor into two separate constructors:

```
template<CopyConstructible T>
class vector {
public:
  vector(size_t n, const T& value);
  where DefaultConstructible<T> vector(size_t n);
};
```

Unfortunately, requiring this transformation for all templates with default argument would make default parameters essentially useless in concept-based template libraries. Instead, we state that default arguments are not type-checked until they are required. We still catch errors at the right time, when the caller attempts to make use of the default argument:

```
template<CopyConstructible T>
vector<T> make_n_vec(size_t n) {
  return vector<T>(n); // error: T has no default constructor
}
```

In essence, this rule is just an extension of the existing rule for default arguments in templates, which states that they will not be instantiated unless required by the caller.

4 Acknowledgments

The effort to introduce concepts into C++ has been shaped by many. The authors of the "Indiana" and "Texas" concepts proposals have had the most direct impact on concepts: Gabriel Dos Reis, Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. Other major contributors to the introduction of concepts in C++ include David Abrahams, Matthew Austern, Mat Marcus, David Musser, Sean Parent, Sibylle Schupp, and Alexander Stepanov. This work was supported by NSF grant EIA-0131354 and by a grant from the Lilly Endowment.

References

[1] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA., 1990.

[2] Douglas Gregor. ConceptGCC: Concept extensions for C++. `http://www.generic-programming.org/software/ConceptGCC`, 2006.

[3] 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.

[4] 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.

[5] 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.

[6] Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine. Concepts for the C++0x Standard Library: Algorithms. Technical Report N2040=06-0110, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[7] Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine. Concepts for the C++0x Standard Library: Approach. Technical Report N2036=06-0106, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[8] Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine. Concepts for the C++0x Standard Library: Introduction. Technical Report N2037=06-0107, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[9] Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine. Concepts for the C++0x Standard Library: Iterators. Technical Report N2039=06-0109, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[10] Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine. Concepts for the C++0x Standard Library: Numerics. Technical Report N2041=06-0111, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[11] Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine. Concepts for the C++0x Standard Library: Utitilies. Technical Report N2038=06-0108, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.

[12] 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.

[13] 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`.

[14] 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`.

[15] 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`.

[16] 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.