Concepts for the C++0x Standard Library: Approach

Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, jewillco, lums}@cs.indiana.edu

Document number: N2036=06-0106
Date: June 23, 2006
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@cs.indiana.edu>

1 Introduction

Concepts are a new language feature proposed for C++0x [2] that provide complete type-checking for template definitions and uses. With concepts, templates can describe requirements on template parameters, e.g., the Iter parameter must meet the requirements of the Input Iterator concept. Users of a template must meet these requirements before the template can be used, and the definition must only use operations on Iter that are part of the requirements in Input Iterator. As the most obvious user-level benefit, concepts can drastically improve the error messages produced by libraries that make heavy use of templates. However, concepts also make it easier to specify and implement template libraries, because they replace a grab bag of *ad hoc* template techniques (traits, tag dispatching, SFINAE, etc.) with a single, coherent, type-safe mechanism.

Concepts were designed with the Standard Library in mind, and both users and implementors of the Standard Library will benefit greatly from the introduction of concepts into the language. On the specification side, what is currently specified through informal descriptions in requirements tables and paragraphs can be specified through C++ code using concepts. Implementors can readily determine that their implementation conforms to the specification because a concept-based compiler checks a template definition against its status requirements. Moreover, many errors that have slipped through the specification process can now be caught by the compiler, such as the infamous vector<**bool**> iterators (a compiler support concepts will reject any attempt to call them Random Access Iterators) or the more-subtle errors resulting from missing requirements in algorithms (e.g., Assignable in unique_copy()). Users will benefit most of all, with a more flexible, simpler, and cleaner implementation of the standard library that produces short, direct error messages.

This document describes how Concepts can be introduced into the C++0x Standard Library. It is accompanied by several other documents, listed below. This particular document details the methodology we have used to integrate concepts into the Standard Library, while the remaining documents provide proposed wording for each chapter of the Standard Library.

How much of the Standard Library will be affected by Concepts? To generate a conservative estimate, search for all occurrences of the term "template" in chapters 17–27 of the current Working Draft for the C++0x standard (N2009). However, even though the scope of this change is large, the risk associated with it is far lower than other changes of this magnitude, because this change introduces improved type safety into the library. Unless otherwise noted, every change described in the proposed wording has been verified with the ConceptGCC compiler [1], which provides a working implementation of concepts along with a concept-enhanced implementation of the Standard Library. Modulo bugs in ConceptGCC or weaknesses in its test suite, we can be certain that the requirements we place on templates will be correct.

This document describes our approach to introducing concepts into the Standard Library. Actual proposed wording is provided in several companion documents, divided by chapter. At present, these documents are not yet complete and will require changes as the concepts language proposal [2] and the working draft (currently N2009) evolve.

- Concepts for the C++0x Standard Library: Chapters 17–20 [**?**]
- Concepts for the C++0x Standard Library: Chaper 24: Iterators [**?**]
- Concepts for the C++0x Standard Library: Chaper 25: Algorithms [**?**]

2 Methodology

This section describes the methodology we employ to introduce concepts into the Standard Library. Our basic approach is to study the specification of a particular template in the Standard Library to determine the requirements it places on its template parameters. We then modify ConceptGCC's Standard Library implementation (libstdc++) by introducing requirements onto that template, then determine if ConceptGCC can type-check the definition of that template. If so, our task is complete; if not, we will study the potential implementations of that template to determine the proper requirements, and iterate the process until we are satisfied that we have described the minimal requirements on the template. We will describe how this process will alter the Standard Library in the following sections.

2.1 Constraining Function Templates

The function templates in the Standard Library rely primarily on convention to convey requirements. For instance, consider the specification of the for_each() algorithm in the current standard:

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function f);
```

1. *Effects*: Applies f to the result of dereferencing every iterator in the range [first, last), starting from first and proceeding to last−1.
2. *Returns*: f
3. *Complexity*: Applies f exactly last− first times.
4. *Remarks*: If f returns a result, the result is ignored.

In this description, the names of the template parameters are significant: **class** InputIterator indicates that any type bound to this type parameter must meet the Input Iterator requirements. With concepts, InputIterator is actually a concept, so the requirement that the first parameter meet the requirements of Input Iterator is stated directly in the declaration, no conventions required. Thus, the specification of for_each() with concepts will look like this:

```
template<InputIterator Iter, typename Function>
  where Callable1<Function, Iter::reference>
  Function for_each(Iter first, Iter last, Function f);
```

1. *Effects*: Applies f to the result of dereferencing every iterator in the range [first, last), starting from first and proceeding to last−1.
2. *Returns*: f
3. *Complexity*: Applies f exactly last− first times.

Here, InputIterator Iter states a requirement that the type Iter must meet the requirements of the InputIterator concept. We have also added a **where** clause, which states that one must be able to call objects of type Function with values of type Iter::reference (i.e., the reference type of the iterator). The compiler will type-check a call to for_each() using these requirements, producing a short, meaningful error message if the requirements are not met. Likewise, the compiler will type-check the definition of for_each() provided by the library, ensuring that the requirements are sufficient to cover the implementation.

We have repeated this same process for every template in the Standard Library, both function and class templates. Most of the changes are direct translations from the existing specification to the concept-based specification, although in some cases we have drastically simplified matters. For instance, consider the existing specification of advance():

1. Since only random access iterators provide + and - operators, the library provides two function templates advance and distance. These function templates use + and - for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

   ```
   template <class InputIterator, class Distance>
     void advance(InputIterator& i, Distance n);
   ```

2. *Requires*: n may be negative only for random access and bidirectional iterators.

3. *Effects*: Increments (or decrements for negative n) iterator reference i by n.

That first paragraph is actually stating that there are three different implementations of advance(), for input, bidirectional, and random access iterators. Within the implementation of the Standard Library, this requires either the use of tag dispatching or SFINAE, complicating library code and confusing curious users. With concepts, we present the three different versions of advance(), and the compiler will select the best version when advance() is invoked:

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

Note that we have also eliminated the Distance template parameter. In some cases, we will clean up artifacts like Distance,[1] but changes such as these will always be accompanied by an editorial comment.

2.2 Requirements Tables → Concepts

The Standard Library currently specifies the requirements that templates place on their template parameters via requirements tables. Requirements tables express the required operations through *valid expressions* that must work with the type, for which return types and semantics are provided.

Concepts contain the same information as requirements tables, but in a form that can be used directly by the compiler. We replace each requirements table with a concept of the same name, translating each valid expression into a signature within the concept. The requirements table and concept for Less Than Comparable are shown below.

Table 29, Less Than Comparable requirements

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

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

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

The two formulations of the Less Than Comparable are essentially identical. However, the concept formulation permits separate type checking of function templates, and leaves much less room for interpretation than the requirements table. Similarly, we can translate the Copy Constructible requirements table into a concept:

[1] Historically, the Distance parameter exists in this declaration only because compilers were unable to handle the use of iterator_traits within the parameter list of advance().

Table 30, Copy Constructible requirements

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

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

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

Note: In the actual library proposal, we eliminate the requirements for the & operator, because they are too restrictive.

As we translate requirements tables into concepts, some refactoring and cleanup is necessary. The following illustrates how the requirements table for Input Iterator can be translated into a concept. It is not the final formulation of the InputIterator concept, but it is representative of the transformation we perform for the non-trivial requirements tables in the standard library.

Table 73, Input Iterator requirements

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

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

```
concept InputIterator<typename X>
   : CopyConstructible<X>, Assignable<X>,
     EqualityComparable<X>
{
  typename difference_type;
  typename value_type;
  typename reference;
  typename pointer;

  where SignedIntegral<difference_type>;
  where Convertible<reference, value_type>;
  where Convertible<pointer, const value_type*>;

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

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

Concepts also take the place of traits (such as iterator_traits) and tag type (such as input_iterator_tag). These constructs will be deprecated in favor of concepts.

3 Conclusion

Introducing concepts into the C++0x Standard Library will affect a large portion of the specification. However, we are following a rigid scheme of translating the existing specification into a concept-based specification, backed up by ConceptGCC's ability to type-check template definitions. The end result will be a more carefully specified Standard Library that is easier to implement and provides a better user experience.

References

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

[2] Douglas Gregor and Bjarne Stroustrup. Concepts. Technical Report N2042=06-0112, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, June 2006.