Dr. Dobb's Software Tools for the Professional Programmer

C++ Concept Checking

Dr. Dobb's Journal June 2001

A better practice for template programming

By Jeremy Siek and Andrew Lumsdaine

Jeremy and Andrew work in the computer science department at Indiana University. They can be contacted at lums@cs.indiana.edu.

Generic programming in C++ is characterized by the use of template parameters to represent abstract data types, that is, "concepts" as the term is used in the SGI STL documentation (see *Generic Programming and the STL*, by M.H. Austern, Addison-Wesley, 1999 and the SGI Implementation of the Standard Template Library, http://www.sgi.com/Technology/STL/). Unfortunately, the flexibility provided by templates generally comes at the expense of interface safety. In this article, we present a technique for reintroducing interface safety into template functions.

Consider Example 1(a). If you have a function that operates on a stack of integers, you could use an object-oriented approach to describe the stack requirements using an abstract base class. A subsequent call *foo(y)* is only valid if the argument *y* bound to *x* is of a type derived from *Stack*. This ensures the type of *y* has at least *push()* and *pop()* member functions so that the calls to these functions inside of *foo()* are guaranteed to be well defined.

With templates, interface safety becomes a much more nebulous issue. For example, suppose you have a function template declared as in Example 1(b). In this case, the compiler lets an object of any type be passed as an argument to *bar()*, whether or not it is a stack (that is, has *push()* and *pop()* member functions). If an incorrect type is used, a compiler error will occur. However, the error will not be caught at the call site of *bar()*, but somewhere deep inside the implementation of *bar()*. What you need is a mechanism for template functions that provides interface safety in a manner reminiscent of abstract base classes. We do this by using several new C++ idioms to specify and then exercise requirements on template parameters. We use the term "concept" to mean a set of requirements and, therefore, call our resulting methodology "concept checking."

A group of us at SGI and the University of Notre Dame began working on concept checking mechanisms more than a year ago, producing the concept checks that are currently in the SGI STL distribution. This work was further developed (see "Concept Checking: Binding Parametric Polymorphism in C++," by J. Siek and A. Lumsdaine, *First Workshop on C++ Template Programming*, October 2000) and is now available in the form of the Boost Concept Checking Library (http://www.boost.org/libs/concept_check/concept_check.htm). In this article we cover two complementary aspects of concept checking — concept checking and concept covering. Concept checking deals with properly expressing the requirements of a concept, verifying that template arguments meet these requirements, and providing appropriate error messages when user-supplied template arguments do not meet these requirements. Concept covering verifies that the stated requirements for a template are valid; that is, that they cover all the ways in which the template parameter is used without introducing unnecessary requirements.

Concepts

A concept is a set of requirements (valid expressions, associated types, semantic invariants, complexity guarantees, and so on) that a type must fulfill to be correctly used as an argument in a call to a generic algorithm. However, C++ has no explicit mechanism for representing concepts — template parameters are merely place holders and express no constraints on the template argument. By convention, template parameters are given names corresponding to the concept that is required, but C++ compilers do not enforce compliance to the concept at the point where the template parameter is bound to an actual type.

Naturally, a compile-time error occurs if a generic algorithm is invoked with a type that does not fulfill at least the syntactic requirements of the concept. However, this error will not reflect that the type did not meet all requirements of the concept per se. Rather, the error may occur at a point where an expression is not valid for the type, or where a presumed associated type is not available. The resulting error messages are largely uninformative and basically impenetrable.

What is required is a mechanism for enforcing "concept safety" at (or close to) the point of instantiation. The Boost Concept Checking Library (BCCL) uses Standard C++ constructs to enforce early concept compliance and provide more informative error messages upon non-compliance. The techniques described here only address the syntactic requirements of concepts (the valid expressions and associated types). These techniques do not address the semantic invariants or complexity guarantees that are also part of concept requirements (although this is an active topic in our research group).

Extended Example

It's happened to us all: One small mistake in using a Standard Library (SL) algorithm and the compiler produces pages of difficult to decipher error messages. Of course this problem is not specific to the SL, but it affects the use of any nontrivial template class or function. Example 2 illustrates such an error. Here we call the *std::stable_sort()* algorithm from the SL, applying it to a linked list.

Attempting to compile this code with GNU C++ produces the compiler error in Example 3. In this case, the fundamental error is that *std::list::iterator* does not model the concept of *RandomAccessIterator*. The list iterator is only bidirectional, not fully random access (as would be a *std::vector::iterator*). Unfortunately, there is nothing in the error message to indicate this to users. The error may be obvious to C++ programmers who have enough experience with template libraries. However, for the uninitiated, there are several reasons why this message would be hard to understand.

- The location of the error. Line 5 of Example 2 is not pointed to by the error message, even though GNU C++ prints up to four levels deep in the instantiation stack.
- There is no textual correlation between the error message and the documented requirements for *std::stable_sort()* and for *RandomAccessIterator*.
- The error message is overly long, listing functions internal to the SL that users do not (and should not) know or care about.
- With so many internal library functions listed in the error message, the user could easily infer the error is due to the library, rather than the user's own code.

Example 4 illustrates what you might expect from a more informative message (and is in fact what the BCCL produces). This message rectifies several of the shortcomings of the previous error messages.

- The location of the error is specified in the error message.
- The message refers explicitly to concepts that the user can look up in the SL documentation *(RandomAccessIterator)*.
- The error message is much shorter and does not reveal internal STL functions.
- The presence of concept_check.hpp and *constraints()* in the error message alerts users that the error lies in the user code, not in the template implementation.

Concept Checking Classes

To check a concept, we employ a special-purpose class (a concept checking class) that ensures that a given type (or set of types) models the given concept. An example of a concept checking class from the BCCL is the *EqualityComparableConcept* class that corresponds to the *EqualityComparable* requirements described in 20.1.1 of the C++ Standard, and to the *EqualityComparable* concept documented in the SGI STL.

```
template <class T>

struct EqualityComparableConcept;
```

The template argument *T* represents the type to be checked. That is, the purpose of *EqualityComparableConcept* is to make sure that the template argument given for *T* models the *EqualityComparable* concept. Each concept checking class has a member function named *constraints(),* which contains the valid expressions for the concept. To check whether some type is *EqualityComparable*, we need to instantiate the concept checking class with the type, then find a way to get the compiler to compile the *constraints()* function without actually executing the function. The BCCL defines two utilities that make this easy: *function_requires()* and BOOST_CLASS_REQUIRES. The *function_requires()* function can be used in function bodies and the BOOST_CLASS_REQUIRES macro can be used inside class bodies.

The *function_requires()* function takes no arguments, but has a template parameter for the concept checking class. This means that the instantiated concept checking class must be given as an explicit template argument; see Listing One(a). With this concept check inserted, if the class *foo* is not a model of *EqualityComparable* (it has not defined operators == and !=), then *function_requires()* will catch the error upfront, instead of allowing the error to occur somewhere inside the template function.

The BOOST_CLASS_REQUIRES macro can be used inside a class definition to check whether some type models a concept; see Listing One(b).

Applications of Concept Checks

One good application of concept checks would be to insert *function_requires()* at the top of *std::stable_sort()* to make sure the template parameter type models *RandomAccessIterator*. In addition, *std::stable_sort()* requires that the value type of the iterators be *LessThanComparable*, so you should also use *function_requires()* to check this; see Listing Two(a).

As an example of using BOOST_CLASS_REQUIRES, look at a concept check that could be added to *std::vector*. One requirement that is placed on the element type is that it must be *Assignable*. You can check this by inserting BOOST_CLASS_REQUIRES at the top of the definition for *std::vector*, as in Listing Two(b).

Although the concept checks are designed for use by generic library implementors, they can also be useful to end users. Sometimes you may not be sure whether some type models a particular concept. This can easily be checked by creating a small program and using *function_requires()* with the type and concept in question. The BCCL file stl_concept_checks.cpp provides an example of applying the concept checks to SL containers.

Implementation

Ideally, we would like to catch (and indicate) the concept violation at the point of instantiation. As mentioned in *Design and Evolution of C++*, by Bjarne Stroustrup (Addison-Wesley, 1994), the error can be caught by exercising all of the requirements needed by the function template. Exactly how the requirements (the valid expressions in particular) are exercised is a tricky issue, because we want the code to be compiled, but not executed. Our approach is to exercise the requirements in a separate function that is assigned to a function pointer. In this case, the compiler instantiates the function but will not actually invoke it. In addition, an optimizing compiler will remove the pointer assignment as dead code (though the run-time overhead added by the assignment would be trivial in any case). It might be conceivable for a compiler to skip the semantic analysis and compilation of the constraint functions in the first place, which would make our function pointer technique ineffective. However, this is unlikely because removal of unnecessary code and functions is typically done in later stages of a compiler. We have successfully used the function pointer technique with GNU C++, Microsoft Visual C++, and several EDG-based compilers (KAI C++, SGI MIPSpro). Listing Four shows how this technique can be applied to the *std::stable_sort()* function.

There is often a large set of requirements that need to be checked, and it would be cumbersome for the library implementor to write constraint functions such as *stable_sort_constraints()* for every public function. Instead, we group sets of valid expressions together, according to the definitions of the corresponding concepts. For each concept, we define a concept checking class template where the template parameter is for the type to be checked. The class contains a *constraints()* member function, which exercises all of the valid expressions of the concept. The objects used in the *constraints()* function, such as *n* and *i*, are declared as data members of the concept checking class; see Listing Five. You can still use the function pointer mechanism to cause instantiation of the constraints function, however, now it will be a member function pointer. To make it easy for library implementors to invoke the concept checks, we wrap the member function pointer mechanism in a function named *function_requires()*. Listing Six(a) shows how to use *function_requires()* to make sure that the iterator is a *RandomAccessIterator*.

The definition of *function_requires()* is as follows: The *Concept* is the concept checking class that has been instantiated with the modeling type. We assign the address of the constraints member function to the function pointer *x*, which causes the instantiation of the constraints function and checking of the concept's valid expressions. We then pass *x* to the *ignore_unused_variable_warning()* function, and wrap everything in a loop to prevent *do-while* collisions; see Listing Six(b).

To check the type parameters of class templates, we provide the BOOST_CLASS_REQUIRES macro that can be used inside the body of a class definition (whereas *function_requires()* can only be used inside of a function body). This macro declares a nested class template, where the template parameter is a function pointer. We then use the nested class *type* in a *typedef*, see Listing Six(c), with the function pointer type of the constraint function as the template argument. We use the *type_var* and concept names in the nested class and typedef names to help prevent name collisions.

In addition, there are versions of BOOST_CLASS_REQUIRES that take more arguments to handle concepts that include interactions between two or more types. BOOST_CLASS_REQUIRES was not used in the implementation of the BCCL concept checking classes because several compilers do not implement template parameters of function pointer type.

Creating Concept Checking Classes

As an example of how to create a concept checking class, look at how to create the corresponding checks for the *RandomAccessIterator* concept. First, as a convention we name the concept checking class after the concept, and add the suffix "Concept." Next, we define a member function named *constraints()* in which we will exercise the valid expressions of the concept. *function_requires()* expects this function's signature to appear exactly as it appears in Listing Three: a void non*const* member function with no parameters.

The first part of the *constraints()* function includes the requirements that correspond to the refinement relationships between *RandomAccessIterator* and the concepts that it builds upon: *BidirectionalIterator* and *LessThanComparable*. We could have instead used BOOST_CLASS_REQUIRES and placed these requirements in the class body, however, BOOST_CLASS_REQUIRES uses C++ features that are less portable.

Next we check that the *iterator_category* of the iterator is either *std::random_access_iterator_tag* or a derived class. After that, we write out some code that corresponds to the valid expressions of the *RandomAccessIterator* concept. *Typedef*s can also be added to enforce the associated types of the concept; see Listing Three.

One potential pitfall in designing concept checking classes is using more expressions in the *constraints()* function than necessary. For example, it is easy to accidentally use the default constructor to create the objects that will be needed in the expressions (and not all concepts require a default constructor). This is the reason we write the *constraints()* function as a member function of a class. The objects involved in the expressions are declared as data members of the class. Since objects of the *constraints()* class template are never instantiated, the default constructor for the concept checking class is never instantiated. Hence, the data members' default constructors are never instantiated (C++ Standard Section 14.7.1 9).

Concept Covering and Archetypes

While it is important to select the minimal requirements (concepts) for the inputs to a component, it is equally important to verify that the chosen concepts cover the algorithm. That is, any possible user error should be caught by the concept checks and not slip through. Concept coverage can be verified through the use of archetype classes. An archetype class is an exact implementation of the interface associated with a particular concept. The run-time behavior of the archetype class is not important — the functions can be left empty. A simple test program can then be compiled with the archetype classes as the inputs to the component. If the program compiles, then one can be sure that the concepts cover the component.

Listing Seven shows the archetype class for the *TrivialIterator* concept. Care must be taken to ensure that the archetype is an exact match to the concept. For example, the concept states that the return type of *operator*()* must be convertible to the value type. It does not state the more stringent requirement that the return type be *T*& or *const T*&. The correct approach is to create an artificial return type that is convertible to *T*, as we have done here with *input_proxy*. The validity of the archetype class test is completely dependent on it being an exact match with the concept, which must be verified by careful (manual) inspection.

Generic algorithms are often tested by being instantiated with a number of common input types. For example, you might apply *std::stable_sort()* with basic pointer types as the iterators. Though appropriate for testing the run-time behavior of the algorithm, this is not helpful for ensuring concept coverage because C++ types never match particular concepts, they often provide much more than the minimal functionality required by any one concept. That is, even though the function template compiles with a given type, the concept requirements may still fall short of covering the function's actual requirements. This is why it is important to compile with archetype classes in addition to testing with common input types.

Listing Eight is an excerpt from the BCCL file stl_concept_covering.cpp that shows how archetypes can be used to check the requirement documentation for *std::stable_sort()*. In this case, it looks like the *CopyConstructible* and *Assignable* requirements were forgotten in the SGI STL documentation (try removing those archetypes). The Boost archetype classes have been designed so that they can be layered. In this example, the value type of the iterator is composed out of two archetypes. In Listing Eight, template parameters named *Base* indicate where the layered archetype can be used.

Requirement Minimization

The process of deciding how to group requirements into concepts and deciding which concepts to use in each algorithm is perhaps the most difficult (yet most important) part of building a generic library. A guiding principle to use during this process is one we call the "requirement minimization principle" — minimize the requirements on the input parameters of a component to increase its reusability.

There is natural tension in this statement. By definition, the input parameters must be used by the component in order for the component to accomplish its task (by "component" we mean a function or class template). The challenge then is to implement the component in such a way that makes the fewest assumptions (the minimum requirements) about the inputs while still accomplishing the task. The traditional notions of abstraction tie in directly to the idea of minimal requirements. The more abstract the input, the fewer the requirements. Thus, concepts are simply the embodiment of abstract data types in C++ template programming. When designing the concepts for some problem domain, it is important to keep in mind their purpose, namely to express the requirements for the input to the components. With respect to the requirement minimization principle, this means we want to minimize concepts.

Minimality in concepts is a property associated with the underlying semantics of the problem domain being represented. In the problem domain of basic containers, requiring traversal in a single direction is a smaller requirement than requiring traversal in both directions (hence the distinction between *ForwardIterator* and *BidirectionalIterator*). The semantic difference can be easily seen in the difference between the set of concrete data structures that have forward iterators versus the set that has bidirectional iterators. For example, singly linked lists would fall in the set of data structures having forward iterators, but not bidirectional iterators. In addition, the set of algorithms that one can implement using only forward iterators is quite different than the set that can be implemented with bidirectional iterators. Because of this, it is important to factor families of requirements into rather fine-grained concepts. For example, the requirements for iterators are factored into the six SL iterator concepts (trivial, input, output, forward, bidirectional, and random access).

The Boost Concept Checking Library

The Boost Concept Checking Library includes concept checking classes for all of the concepts used in the C++ Standard Library (plus a few more). In addition, other Boost libraries include concept checking classes for the concepts that are particular to those libraries. For example, the Boost Graph Library includes checks for the graph concepts and property map concepts that it provides. We encourage you to use concept checking as part and parcel of developing generic algorithms. When writing algorithms that use existing concepts, you should simply reuse the associated concept check. When introducing new concepts, you should develop corresponding concept checks for your own use and for users of your software.

Looking to the Future

Designing, implementing, and verifying concept checks for generic C++ libraries must presently be done manually. As a result, the process is time consuming and (likely to be) error prone. Implementors would benefit greatly if some or all of this process could be automated.

A first step would be to have a tool that statically analyzes a class or function template and records all the kinds of expressions that involve the template parameter types. Such a tool would ease the task of verifying concept coverage. A second step would pattern-match the set of all required expressions against a standard set (or library-defined set) of concepts, thereby summarizing the requirements in terms of concepts. This information could then be used in two ways. First, it could be used to create readable reports for library documentation. Second, it could be used to provide informative compiler error messages without the need to manually insert concept checks. Finally, we remark that there is much work to be done in the general area of generic programming. Outside of C++, there are a number of approaches one could take for designing a language that directly supports concepts.

Acknowledgments

Thanks to Alexander Stepanov for originating the idea of using function pointers to trigger the instantiation of concept checking code. Thanks also to Matthew Austern for establishing the concepts of the SGI STL; Beman Dawes for managing the Boost review of the BCCL; and to all the Boost members for reviewing BCCL. Parts of this work were performed while Jeremy was interning at SGI, and Andrew was on sabbatical at Lawrence Berkeley National Lab. This work was partially funded by NSF grant ACI-9982205.

DDJ

Listing One

(a)

```
// In my library:
template <class T>
void some_function_template(T x)
{
   function_requires< EqualityComparableConcept<T> >();
   // ...
};
// In the user's code:
class foo {
//...
};
int main() {
   foo f;
   some_function_template(f);
   return 0;
}
```

(b)

```
// In my library:
template <class T>
struct some_class_template
{
   BOOST_CLASS_REQUIRES(T, EqualityComparableConcept);
   // ...
};
// In the user's code:
class foo {
   //...
};
int main() {
   some_class_template<foo> glc;
   // ...
   return 0;
}
```

Back to Article

Listing Two

(a)

```
template <class RandomAccessIter>
void stable_sort(RandomAccessIter first, RandomAccessIter last)
{
   function_requires< RandomAccessIteratorConcept<RandomAccessIter> >();
   typedef typename std::iterator_traits<RandomAccessIter>::
                                                 value_type value_type;
   function_requires< LessThanComparableConcept<value_type> >();
   ...
}
```

(b)

```
namespace std {
   template <class T>
   struct vector {
      BOOST_CLASS_REQUIRES(T, AssignableConcept);
      ...
   };
}
```

Back to Article

Listing Three

```
template <class Iter>
struct RandomAccessIteratorConcept
{
   void constraints() {
      function_requires< BidirectionalIteratorConcept<Iter> >();
      function_requires< LessThanComparableConcept<Iter> >();
      function_requires< ConvertibleConcept<
         typename std::iterator_traits<Iter>::iterator_category,
         std::random_access_iterator_tag> >();
      i += n;
      i = i + n; i = n + i;
      i -= n;
      i = i - n;
      n = i - j;
      i[n];
   }
   Iter a, b;
   Iter i, j;
   typename std::iterator_traits<Iter>::difference_type n;
 };
}
```

Back to Article

Listing Four

```
template <class RandomAccessIterator>
void stable_sort_constraints(RandomAccessIterator i)
{
   typename std::iterator_traits<RandomAccessIterator>
      ::difference_type n;
   i += n; // exercise the requirements for RandomAccessIterator
   ...
}
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last)
{
   typedef void (*fptr_type)(RandomAccessIterator);
   fptr_type x = &stable_sort_constraints;
   ...
}
```

Back to Article

Listing Five

```
template <class Iter>
struct RandomAccessIterator_concept
{
   void constraints()
   {
      i += n;
      ...
   }
   typename std::iterator_traits<RandomAccessIterator>
      ::difference_type n;
   Iter i;
   ...
};
```

Back to Article

Listing Six

```
(a)

template <class Iter>
void stable_sort(Iter first, Iter last)
{
   function_requires< RandomAccessIteratorConcept<Iter> >();
   ...
}

(b)

template <class Concept>
void function_requires()
{
   void (Concept::*x)() = BOOST_FPTR Concept::constraints;
   ignore_unused_variable_warning(x);
}

(c)

#define BOOST_CLASS_REQUIRES(type_var, concept) \
   typedef void (concept <type_var>::* func##type_var##concept)(); \
   template <func##type_var##concept _Tp1> \
   struct concept_checking_##type_var##concept { }; \
   typedef concept_checking_##type_var##concept< \
      BOOST_FPTR concept <type_var>::constraints> \
      concept_checking_typedef_##type_var##concept
```

Back to Article

Listing Seven

```
template <class T>
struct input_proxy {
   operator const T&() {
      return static_object<T>::get(); // Get a reference without constructing
   }
};
template <class T>
class trivial_iterator_archetype
{
   typedef trivial_iterator_archetype self;
public:
   trivial_iterator_archetype() { }
   trivial_iterator_archetype(const self&) { }
   self& operator=(const self&) { return *this; }
   friend bool operator==(const self&, const self&) { return true; }
   friend bool operator!=(const self&, const self&) { return true; }
   input_proxy<T> operator*() { return input_proxy<T>(); }
};
namespace std {
   template <class T>
   struct iterator_traits< trivial_iterator_archetype<T> >
   {
      typedef T value_type;
   };
}
```

Back to Article

Listing Eight

```
{
   typedef less_than_comparable_archetype<
      sgi_assignable_archetype<> > ValueType;
   random_access_iterator_archetype<ValueType> ri;
   std::stable_sort(ri, ri);
}
```

Back to Article