Named Requirements for C++0X Concepts

Document #: WG21/N2581 = J16/08-0091
Date: 2008-03-13
Revises: None
Project: Programming Language C++
Reference: ISO/IEC IS 14882:2003(E)
Reply to: Walter E. Brown <wb@fnal.gov>
Chris Jefferson <chris@bubblescope.net>
Alisdair Meredith <alisdair.meredith@codegear.com>
James Widman <widman@gimpel.com>

Contents

As the term is used in mainstream cognitive science and philosophy of mind, a concept is an abstract idea or a mental symbol, typically associated with a corresponding representation in a language or symbology.
— WIKIPEDIA, 2008-03-03

1 Introduction

This paper proposes to augment C++0X with a small new feature, *named requirement*s, in order to achieve considerable reduction in the bulk, the complexity, and the redundancy now needed in the specification of non-trivial constrained templates. Motivated by the experiences of the LWG in specifying the constraints on the well-known algorithms of the Standard Library, rephrasing the requirements via the proposed feature leads to improved ease of specification and comprehension consistent with classical code factorization techniques.

2 Example 1: basic utility of *named requirements*

LWG has observed that concepts with several interrelated requirements today often require duplication of non-trivial requirements in multiple contexts. As a representative example, consider the requirements of the constrained template `std::inner_product`. The template's declaration (reformatted for clarity) was originally articulated as shown in Listing 1:

```
//                                Listing 1
template<InputIterator Iter1, InputIterator Iter2, typename T>
  requires Multiplicable< Iter1::reference
                        , Iter2::reference
                        >
        && Addable< T
                  , Multiplicable< Iter1::reference
                                 , Iter2::reference
                                 >::result_type
                  >
        && Assignable< T
                     , Addable< T
                              , Multiplicable< Iter1::reference
                                             , Iter2::reference
                                             >::result_type
                              >::result_type
                     >
T
  inner_product( Iter1 first1, Iter1 last1, Iter2 first2, T init );
```

Note that the non-trivial `Multiplicable<>` requirement is replicated in its entirety within the `Addable<>` requirement, and that the `Addable<>` requirement is itself replicated in its entirety within the `Assignable<>` requirement. Thus `Multiplicable<>` appears three times in all (as highlighted in red), and `Addable<>` appears twice (as highlighted in brown). Each replication arises in the context of an ever-larger surrounding requirement.

However, employing the proposed *named requirements*, the redundancy and bulk of these requirements can be greatly reduced by the simple technique of naming the requirements and thereafter referring to previously-named requirements instead of replicating them. The resulting code, shown in Listing 2, is vastly simpler to read and comprehend:

```
//                                Listing 2
template<InputIterator Iter1, InputIterator Iter2, typename T>
  requires mult = Multiplicable<Iter1::reference, Iter2:reference>
        && add  = Addable<T, mult::result_type>
        &&        Assignable<T, add::result_type>
T
  inner_product( Iter1 first1, Iter1 last1, Iter2 first2, T init );
```

Note that we do not propose to require that each requirement be named. It is up to the user to determine whether a name should be attached to any given requirement.

3 Example 2: *named requirements* within concept definitions

The code in Listing 3 is adapted from Peter Gottschling's technical report *Fundamental Algebraic Concepts in Concept-Enabled C++* [1]:

[1]TR 638, Indiana University, October 2006.

```
//                                  Listing 3
auto concept BinaryIsoFunction< typename Op, typename Elem >  {
  requires std::Callable<Op, Elem, Elem>;
  requires std::Convertible< std::Callable<Op, Elem, Elem>::result_type
                           , Elem
                           >;
  typename result_type = std::Callable<Op, Elem, Elem>::result_type;
};
```

Note in particular the three occurrences (highlighted in red) of the identical `Callable<>` requirement in the above code. This redundancy could be addressed via *named requirement*s as shown in Listing 4:

```
//                                  Listing 4
auto concept BinaryIsoFunction< typename Op, typename Elem >  {
  requires call = std::Callable< Op, Elem, Elem >;
  requires std::Convertible< call::result_type, Elem >;
  typename result_type = call::result_type;
};
```

4 Example 3: refinement clauses as *named requirements*

Refinement clauses in concept definitions provide another source of potential redundancy. Often a concept refining a non-trivial concept will several times want to refer to the refined concept. For example consider Listing 5 below, in which `Complicated<>` may denote an arbitrarily lengthy concept being refined.

```
//                                  Listing 5
concept Refining< typename T, typename U > : Complicated<T,U>  {
  requires std::HasPlus< Complicated<T,U>::result_type >;
};
```

Note the similarity to the already discussed problem of repeating a complex requirement, as demonstrated in Listing 6.

```
//                                  Listing 6
concept Refining< typename T, typename U >  {
  requires c = Complicated<T,U>;
  requires std::HasPlus< c::result_type >;
};
```

It seems desirable to permit the analogous specification in the context of a refinement clause. Therefore, to simplify the specification as well as for consistency, we propose to allow *named requirement*s in refinement clauses as well as in the body of concepts. The notation, demonstrated in Listing 7, follows naturally from the previous examples.

```
//                                  Listing 7
concept Refining< typename T, typename U > : c = Complicated<T,U>  {
  requires std::HasPlus< c::result_type >;
};
```

5 Design decisions

5.1 Scope

As shown in the above examples, the minimum useful scope for a requirement's name is from the point of its declaration to the end of the clause introduced by `requires`.

As further illustrated by the previous examples, we propose that a requirement's name be additionally injected into the scope of the concept or constrained template in which it arises. Further, a name thus injected into a concept should be visible to any refining concept, and a name thus injected into a constrained class template should be visible to any derived template.

5.2 Syntax

In order to name a requirement, we propose the declarative syntax: *name* = *requirement*. We considered the additional use of a keyword (*e.g.*, `alias` or `using`), but believe this feature needs no additional keyword. Most requirements already follow the `requires` keyword or (in the case of multiple requirements) follow a subsequent `&&`; we believe this is sufficient context to parse the declaration of a requirement's name.

We do not propose this feature for use in connection with requirements expressed in the short form. However, in the interests of completeness (not to mention simplifying the grammar), we do propose to allow a *named requirement* wherever N2571 proposes to allow a requirement to be written. If adopted, we'll be able to write `a = C<T>`:

1. as a *refinement-specifier* (14.9.3 [concept.refine] p1), and
2. as a *requirement* (14.10.1 [temp.req] p1), thus permitting a *named requirement* to be declared within:
   a) a *member-requirement* (9.2 [class.mem])
   b) an *associated-requirement* (14.9.1.3 [concept.req])
   c) the optional *requires-clause* of an *axiom-definition* (14.9.1.4 [concept.axiom]), and
   d) the optional *requires-clause* of a constrained template declaration (14.10).

Finally, as we have no realistic use case for a *named requirement* pack expansion, we propose that any such construct be ill-formed:

```
//                                Listing 8
template<typename ... Ts>
  requires a = C<Ts>...   // error:  requirement aliases may refer only
                          // to requirements that are not pack expansions
void
  f( Ts... );
```

6 Proposed wording

This section's proposed wording is with respect to N2571, and may therefore need minor adjustment should that paper be revised (*e.g.*, so as to cause renumbering). This proposal is purely an extension to N2571; except for very small additions to the underlying grammar, it requires no changes to any existing wording.

The proposed wording is presented in two parts. The *Minimal wording* provides for the definition and use of alias names, the most basic form of the proposed *named requirement*s feature. The *Recommended additional wording* provides for the name injection of these alias names, supplementing their presence in the scope of template/concept parameters as provided under *Minimal wording*) and allowing these names to be used in derived classes and refining concepts.

6.1 Minimal wording

In 14.9.3 [concept.refine], augment the grammar definition of *requirement-specifier*; in 14.10.1 [temp.req], augment the grammar definition of *requirement* and add a grammar definition of *req-alias-def*. Text to be added is indicated in red:

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

requirement :
  req-alias-def$_{opt}$ ::$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*
  ! ::$_{opt}$ *nested-name-specifier*$_{opt}$ *concept-id*

req-alias-def :
  identifier =

Append the following after 14.10.1 [temp.req] p5 (“A negative requirement requires...”):

A *req-alias-def* introduces the *identifier* as a name in the scope of:

- the concept parameters, when the *req-alias-def* appears in a *refinement-specifier* (14.9.3);
- the member declaration, when the *req-alias-def* appears in a *member-requirement* (9.2);
- the enclosing concept, when the *req-alias-def* appears in an *associated-requirement* (14.9.1.3);
- the axiom, when the *req-alias-def* appears in the optional *requires-clause* of an *axiom-definition* (14.9.1.4); or
- the template parameters declared in the *template-parameter-list* immediately before the `requires` keyword, when the *req-alias-def* appears in the optional *requires-clause* of a constrained template declaration.

The name is an alias for the concept instance named in the requirement. It is inserted into its scope immediately when the *req-alias-def* is seen. [*Example:*

```
#include <concepts>

concept C<typename T> {
  typename R;
}

template<typename T>
  requires J = C<T>
J::R  // qualified lookup finds type name R
      // within the concept map archetype C<T'> (3.4.3.3)
  f( T );

auto concept BinaryFunction<typename Op, typename Elem> {
  requires call = std::Callable<Op, Elem, Elem>;
  requires std::Convertible<call::result_type, Elem>;
  typename result_type = call::result_type;
};
```

—end example]

If a *req-alias-def* appears in a *requirement* that is the pattern of a pack expansion, the program is ill-formed. [*Example:*

```
concept C<typename ... Ts> {}

template<typename ... Ts>
  requires a = C<Ts>...   // error:  requirement aliases may refer only
                          // to requirements that are not pack expansions
void
  f( Ts... );
```

—*end example*]

6.2 Recommended additional wording

Continue the above text, appending the following:

When a *req-alias-def* appears in a *refinement-specifier* (14.9.3), the alias name is also inserted into the scope of the concept; this is known as an *injected-requirement-name*.

When a *req-alias-def* appears in the optional *requires-clause* in the definition of a constrained class template or member class of a class template or any further-nested class, the alias name is also inserted into the scope of the class or class template being defined as an *injected-requirement-name*. For purposes of access checking, the *injected-requirement-name* is treated as if it were a public member name. [*Example:*

```
#include <concepts>

concept B<typename T, typename U> {
  typename result_type;
  ...
};

concept C<typename T, typename U> : b = B<T,U> {
  requires std::HasPlus<b::result_type>;
  typename result_type;
  ...
};

template<typename T>
  requires c = C<T,T>
c::b::result_type
  f( const T& );
```

—*end example*]

Also append the following after 3.3.1 [basic.scope.pdecl] p7 (“The point of declaration for an *injected-class-name* . . . ”):

The point of declaration for an *injected-requirement-name* (14.10.1) is immediately following the opening brace of the definition of the concept or class.

7 Summary and conclusion

In this paper, we have proposed a *named requirements* feature as a small extension to the syntax of concept requirements and refinements, and have demonstrated that the proposed feature provides significant benefits in the form of decreased redundancy in the specifications of concepts and constrained templates, with concomitant improvements in the clarity and comprehensibility of the resulting code. We respectfully urge that the proposal and its underlying issues be considered on a time scale consistent with C++0X. However, we recommend that CWG consider this paper only after it has duly approved the basic concepts proposal.

8 Acknowledgments

We thank Bjarne Stroustrup and Douglas Gregor, principal instigators of the concepts work, for their efforts in promulgating what promises to become one of the major features of C++0X.

We also thank the Fermi National Accelerator Laboratory's Computing Division, sponsor of Fermilab's participation in the C++ standards effort, for its past and continuing support of efforts to improve C++ for all our user communities.