Proposed Wording for Scoped Concept Maps

Authors: Jeremy Siek, University of Colorado at Boulder
James Widman, Gimpel Software
Document number: N2414=07-0274
Date: 2007-09-10
Project: Programming Language C++, Core Working Group
Reply-to: Jeremy Siek <jeremy.siek@colorado.edu>

Introduction

This document provides proposed wording for scoped concept maps. Readers unfamiliar with scoped concept maps are encouraged to read the proposal for scoped concept maps [1]. This document assumes that the revised wording for concepts, paper number N2398=07-0258, has been applied to the working draft, and suggests further changes to the working draft. The changes with respec to N2398=07-0258 are summarized as follows:

— Added concept map names (3,3.4).

— Added using declarations and directives for concept maps (7.3.3).

— Added *nested-name-specifier* in front of *concept-id* in the grammar rule for *concept-map-definition* (14.9.2).

— Removed restriction that concept maps be defined in the same namespace as their concept (14.9.2).

— Changed the *concept-id* requirement to say that it finds concept maps via name lookup (14.9.3).

— Described how concept map lookup works during instantiation on constrained templates (14.9.4).

Typographical conventions

Within the proposed wording, text that has been added will be presented in blue and underlined when possible. Text that has been removed will be presented in red,with strike-through when possible.

Purely editorial comments will be written in a separate, shaded box.

The wording in this document is based on the latest C++0x draft, currently N2284. We have done our best to synchronize section, paragraph, and table numbers with N2284. However, we have omitted many sections (those that did not require any changes), leaving some dangling references in the final document. These references will be resolved automatically when the LaTeX source of this proposal is merged with the LaTeX source of the working paper.

Chapter 3 Basic concepts [basic]

6 Some names denote types, classes, concepts, concept maps, enumerations, or templates. In general, it is necessary to determine whether or not a name denotes one of these entities before parsing the program that contains it. The process that determines this is called *name lookup* (3.4).

3.4 Name lookup [basic.lookup]

1 The name lookup rules apply uniformly to all names (including *typedef-names* (**??**), *namespace-names* (**??**), *concept-names* (**??**), *concept-map-names* (14.9.2) and *class-names* (**??**)) wherever the grammar allows such names in the context discussed by a particular rule. Name lookup associates the use of a name with a declaration (**??**) of that name. Name lookup shall find an unambiguous declaration for the name (see **??**). Name lookup may associate more than one declaration with a name if it finds the name to be a function name; the declarations are said to form a set of overloaded functions (**??**). Overload resolution (**??**) takes place after name lookup has succeeded. The access rules (clause **??**) are considered only once name lookup and function overload resolution (if applicable) have succeeded. Only after name lookup, function overload resolution (if applicable) and access checking have succeeded are the attributes introduced by the name's declaration used further in expression processing (clause **??**).

Chapter 7 Declarations [dcl.dcl]

7.3.3 The `using` declaration [namespace.udecl]

1 A *using-declaration* introduces a name into the declarative region in which the *using-declaration* appears. That name is a synonym for the name of some entity declared elsewhere.

using-declaration:
`using` `typename`$_{opt}$ `::`$_{opt}$ *nested-name-specifier unqualified-id* `;`
`using` `::` *unqualified-id* `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept_map` *nested-name-specifier*$_{opt}$ *concept-id* `;`
`using` *nested-name-specifier*$_{opt}$ `concept_map` *concept-name*$_{opt}$ `;`
`using` `::`$_{opt}$ *nested-name-specifier*$_{opt}$ `concept` *concept-name* `;`

21 A *using-declaration* for a concept map is an alias to the concept map that matches (**??**) the *concept-id* from the specified namespace. [*Example:*

```
namespace N1 {
  concept C<typename T> { };
}
namespace N2 {
  concept_map N1::C<int> { }
  template<typename T> concept_map N1::C<T*> { }
}
namespace N3 {
  using N2::concept_map N1::C<int>;
  using N2::concept_map N1::C<int*>;
}
```

— *end example*]

22 A *using-declaration* for a concept map that specifies a *concept-name* (and not a *concept-id*) brings all of the concept maps and concept map templates from the specified namespace for the given concept into the scope in which the *using-declaration* appears. [*Example:*

```
namespace N1 {
  concept C<typename T> { };
  template<C T> void f(T) { }
}
namespace N2 {
  concept_map N1::C<int> { }
  template<typename T> concept_map N1::C<T*> { }
}
```

```
namespace N3 {
  using N2::concept_map N1::C;
  void g() {
    f(1);
    f(new int);
  }
}
```

— *end example*]

23 If no concept is specified in the concept map using declaration, all concept maps from the specified namespace are brought into scope.

[*Note:* A *using-directive* for a namespace brings the concept maps of that namespace into scope, just like other entities. — *end note*] [*Example:*

```
namespace N1 {
  concept C<typename T> { }
}
namespace N2 {
  concept_map N1::C<int> { }
}
namespace N3 {
  using namespace N2;

  template<N1::C T> void foo(T) { };

  void bar() {
    foo(17); // ok, finds the concept map from N2
  }
}
```

— *end example*]

Chapter 14 Templates [temp]

14.9.1 Concept definitions [concept.def]

3 Each concept has an associated *concept-map-name*, which is an implementation-defined identifier unique to the concept. [*Note:* The concept map name of a concept is not introduced into any scope until a concept map definition is seen. — *end note*]

14.9.2 Concept maps [concept.map]

1 The grammar for a *concept-map-definition* is:

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

concept-map-member-specification:
concept-map-member concept-map-member-specification$_{opt}$

concept-map-member:
simple-declaration
function-definition
template-declaration

2 Concept maps describe how a set of template arguments meet the requirements stated in the body of a concept definition (14.9.1). Whenever a constrained template (**??**) is named, there shall be a concept map corresponding to each *concept-id* requirement in the requirements clause (14.9.3). This concept map may be written explicitly (14.9.2), instantiated from a concept map template (**??**), or generated implicitly (**??**). The concept map name (14.9.1) is inserted into the scope in which the concept map or concept map template (**??**) is defined immediately after the *concept-id* is seen.

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

14.9.3 Template requirements [temp.req]

3 A *concept-id* requirement requires that there is a most specific concept map or concept instance according to concept map matching (**??**) among the concept maps or concept instances that are found by name lookup (3.4) for the *concept-map-name* determined by the concept in the *concept-id*.

14.9.4 Instantiation of constrained templates [temp.constrained.inst]

2 Instantiation of a constrained template also replaces each concept instance associated with the template requirements with the concept map that satisfied the requirement. [*Note:* Concept members that had resolved to members of the concept instance now refer to members of the corresponding concept maps. — *end note*]

3 If a *concept-id* appears (directly or indirectly) multiple times in the requirements of a constrained template, the program is ill-formed if the concept maps used to satisfy the multiple occurences of the *concept-id* are not the same concept map. [*Example:*

```
concept A<typename T> { }
concept B<typename T> {
  typename X;
  requires A<X>;
}
concept C<typename T> {
  typename X;
  requires A<X>;
}
namespace N1 {
  concept_map A<int> { }  // #1
  concept_map B<int> { }  // uses #1 to satisfy the requirement for A<int>
}
namespace N2 {
  concept_map A<int> { }  // #2
  concept_map C<int> { }  // uses #2 to satisfy the requirement for A<int>
}
tempate<typename T> requires B<T> && C<T>
struct S { };
using N1::concept_map B<int>;
using N2::concept_map C<int>;
S<int> s; // ill-formed, two different concept maps for A<int>, #1 and #2
```

— end example]

The point of the above restriction is to allow the implementation to consider requirements as a mathematical set, where duplicates are discarded. Because of the above restriction, it doesn't matter which concept maps for a given concept-id are discarded because they are all the same.

6 The concept maps that are used in this partial ordering include the concept maps that 1) have replaced the concept instances used in the first partial ordering, 2) are in the same namespace as the concept maps from 1), or 3) are in the associated namespaces of the concept map arguments for the concept maps from 1). [*Example:*

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

namespace N1 {
  concept_map C<int> { }
  concept_map D<int> { }
}
namespace N2 {
  template<C T> void f(T);                 // #1
  template<C T> requires D<T> void f(T);   // #2
  template<C T> void g(T x) {
    f(x);
  }
  using N1::concept_map C<int>;
```

```
    void h() {
      g(1); // inside, g, the call to f goes to #2
    }
  }
```

— *end example*]

The point of including 3) above is to handle the common case where the concept maps are defined in the same namespace as the types that they are mapping.

Bibliography

[1] J. Siek. Scoped concept maps. Technical Report N2098=06-0168, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, September 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2098.pdf`.