Scoped Concept Maps

N2098=06-0168

Jeremy Siek

September 10, 2006

Abstract

In this paper I point out a disadvantage of a design choice concerning the "concept map" feature in the concepts proposal (N2042) and I propose an alternative design choice called "scoped concept maps". The disadvantage of the current design is that One-Definition-Rule (ODR) violations may occur when a programmer tries to use libraries that were developed independently of one another, which is quite common in the component-oriented world of modern software development.

1 The Problem

Consider the following scenario. A software developer named John has developed the following library:

```
namespace John {
  template<class T> class fast_vector { ... };
}
```

Independently, a developer named Alex has created some new algorithms and concepts.

```
namespace Alex {
  concept Fooable<typename T> {
    void foo(T);
  };

  template<Fooable T>
  void algorithm_bar(T x);
}
```

Another library developer named Fred would like to use John's fast vector with Alex's algorithms, so he defines a concept map to make fast_vector Fooable. As per 3.3.2 of N2042, Fred must place this concept map in namespace Alex, the namespace where Fooable was defined.

```
// Fred.hpp
namespace Alex {
  concept_map Fooable<John::fast_vector> {
    void foo(const John::fast_vector& v) { v.push_back(4) }
  };
}
namespace Fred {
  void algo() {
    John::fast_vector v;
    Alex::algorithm_bar(v);
  }
}
```

John and Alex's libraries continue to grow in popularity, so another library developer named Suzy decides to use them together, and also defines a concept map for fast_vector. Suzy does not know about Fred's library.

```
// Suzy.hpp
namespace Alex {
  concept_map Fooable<John::fast_vector> {
    void foo(const John::fast_vector& v) { v.push_back(7); }
  };
}
namespace Suzy {
  void algo() {
    John::fast_vector v;
    Alex::algorithm_bar(v);
  }
}
```

Note that Fred and Suzy's need for a Fooable concept map for fast_vector is purely for ***internal*** implementation reasons. However, they have both been forced to add a definition to a namespace ***external*** to their own.

An application developer named Zack would like to use both Suzy and Fred's libraries. However, as soon as Zack includes both Fred.hpp and Suzy.hpp, there's an ODR violation because there are two definitions for the same concept map.

```
// Zack.cpp
#include "Suzy.hpp"
#include "Fred.hpp" // ODR violation!

int main() {
  Suzy::algo();
  Fred::algo();
}
```

Zack does not have access to the source code of Suzy or Fred's libraries, so he can't change their code. Even if he did have access to their code, he would not want to change it himself because that would mean patching their code for every new version. Zack instead tries to get Suzy or Fred to change their libraries, but they are just too busy to be bothered.

At this point, the reader may be thinking, well, we run into ODR violations all the time, what makes this different? Why do we need to solve this problem? Normally, library authors can take measures to prevent ODR violations, for example, by placing their struct definitions inside their library's namespace. In this case, Suzy and Fred had no choice; they are not allowed to put the concept maps in their own namespace, but must put the concept maps in namespace Alex.

2 Scoped Concept Maps

The solution we propose is to lift the restriction that concept maps must be defined in the same namespace as their concept and change the rules for concept map lookup. Currently, concept map lookup only considers one namespace, the namespace of the concept. Instead, we propose that concept lookup be lexical, starting in the scope where the lookup is triggered, and proceeding to enclosing scopes.

A concept map definition that can not be found lexically (for example, the concept map is in another namespace) is not considered unless the concept map is imported with a **using** declaration. Thus, we also propose to extend **using** declarations so that they may refer to concept maps.

To revisit the scenario, Fred and Suzy and may now define their concept maps in their own namespaces.

```
// Fred.hpp
namespace Fred {
  concept_map Alex::Fooable<John::fast_vector> {
    void foo(const John::fast_vector& v) { ... }
  };
  void algo() {
    John::fast_vector v;
    Alex::algorithm_bar(v); // This call uses Fred's concept map
  }
}

// Suzy.hpp
namespace Suzy {
  concept_map Alex::Fooable<John::fast_vector> {
    void foo(const John::fast_vector& v) { ... }
  };
  void algo() {
    John::fast_vector v;
    Alex::algorithm_bar(v); // This call uses Suzy's concept map
  }
}
```

Now Zack can use both Suzy and Fred's libraries without causing an ODR violation.

```
// Zack.cpp
#include "Suzy.hpp"
#include "Fred.hpp" // OK!

int main() {
  Suzy::algo();
  Fred::algo();
}
```

3 Compilation of Scoped Concept Maps

The namespace restriction on concept maps in the current concept proposal (N2042) is motivated by a particular implementation approach. Concepts can be implemented as templates and concept maps can be implemented as template specializations. Because template specializations must be defined in the same namespace as the primary template, concept maps must be defined in the same namespace as their concept.

Scoped concept maps will require a different implementation approach. We outline one such approach here, which we call the "static map passing approach". The basic idea is that a concept map is translated into a struct and every constrained template is translated into a template that has an extra template type parameter for each constraint. When a constrained template is used, the compiler determines (by lexical lookup) which concept map to use, and uses the corresponding struct as the argument to the template's extra constraint parameter. For example, the program

```
concept Comparable<typename T> {
  bool operator<(T,T);
};
struct bar { bar(int x) : x(x) {} int x; };
bool operator<(const bar& b1, const bar& b2) { ... }
concept_map Comparable<bar> { };

template<Comparable T>
void foo(T x) { x < x; }

int main() { foo(bar(1)); }
```

would be translated to something like:

```
struct bar { bar(int x) : x(x) {} int x; };
bool operator<(const bar& b1, const bar& b2) { ... }
struct Comparable_bar {
  bool operator<(const bar& b1, const bar& b2) { return ::operator<(b1,b2); }
};

template<typename Comparable_T, typename T>
void foo(T x) { Comparable_T::operator<(x, x); }

int main() { foo<Comparable_bar>(bar(1)); }
```

4 Conclusion

The concepts proposal (N2042) requires concept maps to be defined in the same namespace as the corresponding concept. This restriction forces third party libraries to place possibly conflicting concept maps in the same namespace, thereby risking violations of the One-Definition-Rule. Here we propose lifting that restriction, allowing concept maps to be defined in any namespace, and we propose changing the concept map lookup rules to be lexical. With this change, third party libraries can define concept maps within their own namespace and completely avoid any chance of ODR violations, thereby making the life of Zack, the application programmer, much easier for he is free to use all the independently developed libraries without risk of hidden conflicts.