Document Number: J16/05-0148 = WG21 N1888
Date: 1 October, 2005
Reply to: William M. Miller
Edison Design Group, Inc.
`wmm@edg.com`

Defining Members of Explicit Specializations

I. Background and Analysis

When a class template or a member class of a class template is explicitly specialized, it's a reasonable expectation that the static data members, member functions, and nested classes of the explicit specialization can be defined at namespace scope, just like the members of other classes and class templates. It appears that the only explicit guidance given by the Standard on how these definitions are to be written is found in 14.7.3¶4:

> A member of an explicitly specialized class is not implicitly instantiated from the member declaration of the class template; instead, the member of the class template specialization shall itself be explicitly defined. In this case, the definition of the class template explicit specialization shall be in scope at the point of declaration of the explicit specialization of the member. The definition of an explicitly specialized class is unrelated to the definition of a generated specialization. That is, its members need not have the same names, types, etc. as the members of a generated specialization. Definitions of members of an explicitly specialized class are defined in the same manner as members of normal classes, and not using the explicit specialization syntax. [*Example:*

```
template <class T> struct A {
  void f(T ) { /* ... */ }
};

template <> struct A<int> {
  void f(int);
};

void h()
{
  A<int> a;
  a.f(16);       // A<int>::f must be defined somewhere
}

// explicit specialization syntax not used for a member of
// explicitly specialized class template specialization
void A<int>::f () { /* ... */ }
```

> *–end example*]

It is not immediately clear from this paragraph exactly which kinds of explicit specializations are in view in this prohibition of "using the explicit specialization syntax." There are three distinct kinds of explicit specializations that can have members, i.e., that might plausibly be the object of this specification:

1. An explicit specialization of a class member of a class template. For example,

   ```
   template <typename T> struct S {
      struct X { };
   };

   template<>
   struct S<int>::X {
      void f();
   };
   ```

2. An explicit specialization of a class template (which might be a non-member, a member of a class, or a member of a class template) that is fully specialized, i.e., specialized as a class. For example,

   ```
   template <typename T1> struct S {
      template <typename T2> struct X { };
   };

   template<> template<>
   struct S<int>::X<int> {
      void f();
   };
   ```

3. An explicit specialization of a class template member of a class template that "remains unspecialized" (14.7.3¶18), i.e., that is specialized as a class template. For example:

   ```
   template <typename T1> struct S {
      template <typename T2> struct X { };
   };

   template<>
   template <typename T> struct S<int>::X {
      void f();
   };
   ```

The question that must be answered for each of these examples is whether 14.7.3¶4 specifies how the member function `X::f()` is to be defined. This question primarily depends on the meaning of "explicitly specialized class" ("Definitions of members of an explicitly specialized class... [do not use] the explicit specialization syntax").

One clue is the example that is given, which is a class template specialized as a class. Another is the fact that the first sentence of the paragraph uses the terms "explicitly specialized class" and "class template specialization" as synonyms. From this we can conclude that an "explicitly specialized class" is not a member class that has been explicitly specialized (example #1), because it is not a "class template specialization" – i.e., the entity that is specialized is a class, not a class template. Similarly, it would appear that an "explicitly specialized class" cannot be a member class template that has been explicitly specialized as a template (example #3), because neither the entity being specialized nor the result of the specialization is a "class." This leaves only example

#2 affected by this paragraph; while the example in the Standard shows a non-member class template, it seems reasonable to conclude that “an explicitly specialized class” should also apply to specializations of class templates that are members of classes or class templates. For example, `X::f()` in example #2 should presumably be defined as

```
    void S<int>::X<int>::f() { }
```

i.e., without “the explicit specialization syntax” (`template<>` prefix).

Given that the Standard does not explicitly specify how to handle examples #1 and #3, are there any inferences that can be drawn from what the Standard does say? Such conclusions must always be made cautiously, but a few extrapolations seem possible.

First, all three of the examples above are essentially similar in structure, differing only with respect to whether the entity being specialized and the result of the specialization are a class or a class template. Given that the Standard explicitly says that one of the three is to be defined without `template<>`, one’s initial inclination might be to conclude that the other two examples ought also to be defined without `template<>`.

This conclusion is supported by the observation that the `template<>` prefix is referred to as “the explicit specialization syntax,” and the definition of a member of an explicit specialization is not itself an explicit specialization. In fact, nowhere in the Standard is there any suggestion that `template<>` is to be used for anything other than an explicit specialization, neither in normative text nor in non-normative notes or examples.

Yet another way of approaching this question is based on the observation that the difference between the first two examples and example #3 is that the earlier specializations are classes, while in #3 it is a class template. With ordinary classes and class templates, i.e., not the result of explicit specialization, the difference in member definitions is straightforward: a member of a class template is defined just like a member of a class except that the declaration is prefixed by a template parameter clause and the *nested-name-specifier* is a *template-id* naming the template parameter(s) instead of an *identifier*. That is, all that is needed to transform the definition of a member of a class into the definition of a member of a class template is to add the underlined portions as follows:

```
    template <typename T>
    void C<T>::f() { }
```

Example #3 relates to the other two examples in exactly the same way that a class template relates to a class: the result of the specialization is a class template rather than a class. Analogy would therefore suggest that the parallel transformation for the definition of `f()` in example #3 would be:

```
    template <typename T>
    void S<int>::X<T>::f() { }
```

All of these considerations seem to lead to the conclusion that the explicit specialization syntax, `template<>`, should be used only for explicit specializations and never for defining members of explicit specializations.

II. Implementation Survey

In order to ascertain how different implementations handle these definitions, several participants on the Core Language Working Group email reflector compiled code samples, both with and without a `template<>` prefix in the member function definitions, and reported their results. The examples were as follows (Sample A corresponds with Example #1, while Sample B reflects Example #3):

```
// Sample A:

template <typename T1> struct A {
   struct C {
      void f() { }
   };
};

template <>
struct A<int>::C {
   void f();
};

// template <>
void A<int>::C::f() { }

// Sample B:

template <typename T1> struct B {
   template <typename T2> struct C {
      void f() { }
   };
};

template <>
template <typename T2> struct B<int>::C {
   void f();
};

// template <>
template <typename T2> void B<int>::C<T2>::f() { }
```

The results of the survey were as follows:

	without `template<>`		**with `template<>`**	
Implementation	**Sample A**	**Sample B**	**Sample A**	**Sample B**
Compaq C++ 6.5	accept	reject	reject	accept
Digital Mars	reject	accept	accept	accept
EDG 3.3	accept	reject	reject	accept
EDG 3.4	accept	reject	reject	accept

EDG 3.5	accept	reject	reject	accept
EDG 3.6	accept	reject	reject	accept
g++ 2.95.2	accept	reject	reject	reject
g++ 3.0	accept	reject	reject	reject
g++ 3.3.1	accept	reject	reject	accept
g++ 3.4.4	accept	reject	reject	accept
g++ 4.0.0	reject	reject	reject	accept
g++ 4.0.1	reject	reject	reject	accept
HP aCC 3.27	reject	reject	reject	reject
HP aCC 5.57	reject	reject	reject	reject
HP aCC 6.0	accept	reject	reject	accept
IBM VAC++ 5.0	accept	accept	accept	reject
IBM VAC++ 6.0	accept	accept	accept	reject
IBM XL C/C++ 7.0	accept	accept	accept	accept
Metrowerks 3.2.6	accept	reject	accept	accept
Metrowerks 4.0.1	accept	reject	accept	accept
MIPSpro 7.41	accept	reject	reject	accept
MSVC++ 7.1	accept	reject	accept	accept
MSVC++ 8.0 beta	accept	reject	accept	accept
Sun 5.3	accept	reject	accept	reject
Sun 5.4	accept	reject	accept	reject
Sun 5.5	accept	reject	reject	reject
Sun 5.6	accept	reject	reject	reject
Sun 5.7	accept	reject	reject	reject
Sun Studio 11	accept	reject	reject	reject
Sun (unreleased)	accept	reject	reject	reject

While it is clear that there is a good deal of variability among the implementations regarding the handling of these examples, some trends are apparent. Among current versions, only two implementations (Digital Mars and IBM) accept Sample B without the `template<>` prefix, while only two (Digital Mars and g++) reject Sample A without `template<>`. The only implementation (Sun) that rejects Sample B with `template<>` also rejects it without `template<>`, so it does not shed much light on whether the prefix should or should not be included; otherwise, all implementations accept Sample B with `template<>`. The most variability among implementations is in the handling of Sample A with `template<>`, with some rejecting and some accepting it.

III. Summary

There are three distinct cases in which an explicitly-specialized entity might declare members that can be defined in namespace scope; however, the current wording of the Standard only specifies the syntax to be used in such member definitions for one of the three cases. Arguments from consistency and analogy can be made that the other two cases ought to follow the same pattern as the one that is explicitly specified, namely, that `template<>` should not be used in the definition of a member of an explicit specialization.

Current implementation practice, however, varies from this analysis. Most implementations require `template<>` on the definition of a member of a member class template specialized as a template (Sample B), and several accept `template<>` on the definition of a member of a specialized member class of a class template (Sample A). However, there is a substantial amount of variability among implementations, and the Standard should make clear if these namespace-scope definitions are permitted and, if so, the syntax that they require.