X3J16/97-0011
WG21/N1049

Template Issues and Proposed Resolutions
Revision 17

John H. Spicer
Edison Design Group, Inc.
jhs@edg.com

January 28, 1996

Revision History

Version 16 (96-0158/N0976) – July 17, 1996: Distributed in the post-Stockholm mailing. Reflects decisions made in Stockholm.
Version 17 (97-0011/N1049) – January 28, 1997: Distributed in the pre-Nashua mailing. Reflects decisions made in Kona and contains new issues.

Summary of Issues

Other Issues

6.46 What are the rules used to determine whether expressions involving nontype template parameters are equivalent?

6.47 When are friend functions defined in class templates evaluated? (closed in version 16)

6.48 Are template friend declarations permitted in local classes? (closed in version 16)

6.49 Where are partial specializations allowed?

6.50 Clarification of the interaction of friend declarations and partial specializations.

6.52 Clarification of ordering rules for nontype arguments in partial specializations.

6.53 Clarification of rules for partial specializations of member class templates.

6.54 Array/function decay in template parameter/argument lists.

Member Template Issues

8.3 Can a member function template be used as a copy constructor or copy assignment operator? (closed in version 17)

8.10 What kind of entity can appear in a template friend declaration?

Other Issues

6.46 Question: What are the rules used to determine whether expressions involving nontype template parameters are equivalent?

Status: Open

A template may be declared in one (or more) translation unit(s) and defined in still another. Because such declarations may involve expressions containing nontype parameters, rules are needed to determine when one such declaration in one translation unit is considered to match another declaration in a different translation unit.

Nontype template parameters cannot be deduced from function parameters in which they are used in expressions, but they can be used in nondeduced contexts (such as return types) and when explicitly specified.

file1.c:

```
template <int I> struct A {};
template <int I, int J> A<I+J> operator +(A<I>, A<J>);
template <int I, int J, class T>
void f(A<I>, A<I*2>, A<(I + J + sizeof(T));

int main()
{
        A<1> a1;  A<2> a2;  A<3> a3;  A<7> a7;
        a3 = a1 + a2;

        f<1,2,int>(a1, a2, a7);
}
```

file2.c:

```
template <int I> struct A {};
template <int I, int J> A<J+I> operator +(A<I>, A<J>);  // error
template <int I, int J, class T>
void f(A<I>, A<I*2>, A<(sizeof(T) + (I + J));  // error
```

Answer: Expressions involving nontype template parameters are compared using an ODR-like rule (can the ODR wording be extended to cover this case?). That is, the tokens that make up the expression must be identical, and the names of entities must refer to the same entities in each translation unit. If two templates are considered to "potentially equivalent", but violate this rule, the results are undefined (or should it be ill-formed with no diagnostic required?) Two templates of a given name in a given scope are considered "potentially equivalent" if they have identical template parameter lists and if, for every given set of template arguments, they result in the generation of functions with the same type.

In other words, function templates that are ODR-identical are guaranteed to refer to the same entity. Function templates that are not "potentially equivalent" are guaranteed to refer to different entities. And function templates that are potentially equivalent render a program undefined (ill-formed, no diagnostic required?)

```
// Guaranteed to be the same
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+10>);

// Guaranteed to be the different
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+11>);

// Undefined whether these two declarations refer to the same template
template <int I> void f(A<I>, A<I+10>);
template <int I> void f(A<I>, A<I+1+2+3+4>);
```

Version added: 15
Version updated: 15

6.49 Question: Where are partial specializations allowed?

Status: Open

Answer: A partial specialization must be declared in the class or namespace in which it is a member. Once declared, it may later be defined outside of the class or namespace.

Version added: 17
Version updated: 17

6.50 Question: Clarification of the interaction of friend declarations and partial specializations

Status: Open

Answer: A template friend class declaration indicates that all instances of that template are friends of the given class regardless of whether those instances are generated from the primary template, a partial specialization, or a full specialization (i.e., explicit specialization).

```
class X {
        template <class T> friend struct A;
        class Y {};
};

template <class T> struct A { X::Y ab; };  // okay
template <class T> struct A<T*> { X::Y ab; };  // okay
```

Consequently, a friend declaration is not allowed to declare a partial specialization:

```
template <class T> struct A {};
template <class T> struct A<T*> {};

class X {
        template <class T> friend struct A<T*>;  // not allowed
};
```

Version added: 17
Version updated: 17

6.51 Clarification of nontype dependency rule in partial specializations

Status: Open

14.5.4 [temp.class.spec] paragraph 5, bullet 2 was editorially changed from

- The type of a specialized nontype argument shall not depend on another type parameter of the specialization.

to

- The type of a specialized argument shall not depend on another type parameter of the specialization.

This rule was intended to prohibit examples such as:

```
template <class T, T t> struct A;
template <class T> struct A<T, 1>;
```

There are two problems with the change that was made. The first problem is that the "specialized" vs. "nonspecialized" distinction only exists for nontype arguments, not for type arguments, so it is impossible to know what a "specialized" type argument might be. But, assuming that a "specialized" type argument is anything other than a type parameter name, the wording change prohibits a large class of partial specializations that should be permitted, such as:

```
template <class T1, class T2> struct A {};
template <class T1> struct A<T1, T1*> {};
```

Answer: The original wording should be restored.

Version added: 17
Version updated: 17

6.52 Clarification of ordering rules for nontype arguments in partial specializations.

Status: Open

A mistake in the original description of the partial ordering rules for nontype arguments has resulted in an unnecessary complication of the partial ordering rules, a gratuitous difference in the partial ordering rules for classes and functions, and a needless restriction in the kinds of partial specialization that can be done.

The fundamental mechanism used to determine ordering of partial specializations is the template argument deduction mechanism. All of the other rules relating to partial specializations are really clarifications of the implications of the argument deduction process. For example, the restriction that nontype arguments cannot be used in expressions in partial specializations is derived from the fact that it is not generally possible to deduce values from expressions.

Recall that the original example, which is prohibited by the WP rules, is:

```
template<int I, int J, class T> class X {};            // #1
template<int I, int J>          class X<I, J, int> {}; // #2
template<int I>                 class X<I, I, int> {}; // #3
```

But the same example rewritten as a set of overloaded function templates is permitted, and calls made using this set of templates will select the appropriate template.

```
template <int I, int J, class T> void f(X<I, J, T>);   // #1
template <int I, int J>          void f(X<I, J, int>); // #2
template <int I>                 void f(X<I, I, int>); // #3
```

Answer:

The WP should be modified as suggested by Fergus Henderson in `c++std-core-7283`:

In

```
| 14.5.4.2  Partial ordering of class template        [temp.class.order]
|           specializations
```

replace

```
| 1 For two class template partial specializations, the first is at  least
|   as specialized as the second if:
|
|   --the  type  arguments  of  the  first template's argument list are at
|     least as specialized as those of the second template's argument list
|     using the ordering rules for function templates (_temp.func.order_),
|     and
|
|   --each non-type argument of the first template's argument list  is  at
|     least as specialized as that of the second template's argument list.
|
| 2 A non-type argument is at least as  specialized  as  another  non-type
|   argument if:
|
|   --both are formal arguments, or
|
|   --the first is a value and the second is a formal argument, or
|
|   --both are the same value.
```

with

```
| 1 For two class template partial specializations, the first is at
|   least as specialized as the second if the arguments of the first
|   template's argument list are at least as specialized as those of
|   the second template's argument list using the ordering rules for
|   function templates (_temp.func.order_).
```

Version added: 17
Version updated: 17

6.53 Clarification of rules for partial specializations of member class templates.

Status: Open

1. When a member template of a class template is partially specialized, the partial specializations apply to all instances generated from the enclosing class template.
2. When the primary template is specialized for a given instance of the enclosing class, none of the partial specializations of the original primary template are carried over.

```
template <class T> struct A {
        template <class T2> struct B {};  // #1
        template <class T2> struct B<T2*> {};  // #2
};

template <> template <class T2> struct A<short>::B {};  // #3

A<char>::B<int*> abcip;  // uses #2
A<short>::B<int*> absip;  // uses #3
```

3. When the primary template is not specialized, the individual partial specializations of the primary template may be specialized (but additional partial specializations cannot be added).

```
template <class T> struct A {
        template <class T2> struct B {};  // #1
        template <class T2> struct B<T2*> {};  // #2
};

template <> template <class T2> struct A<short>::B<T2*> {};  // #3

A<short>::B<int> absi;  // uses #1
A<short>::B<int*> absip;  // uses #3
```

Version added: 17
Version updated: 17

6.54 Array/function decay in template parameter/argument lists.

Status: Open

Sean Corfield, in `c++std-ext-3734` raised the issue of whether the handling of nontype array parameters in the WP is correct.

The WP specifies that type decay (array to pointer and function to pointer) does not occur for template nontype parameters (this was specified by issue 2.12 in revision 6 of this paper), but seems to suggest that the decay takes place on the template argument side.

The decay should either take place on both sides or on neither side. As things now stand, you can declare a nontype parameter of array type, but you can't actually supply an array argument.

After investigating what a number of existing compilers do in this situation, my recommendation is that they decay should occur on both the parameter and argument side. EDG,

g++, Microsoft, Sun, and Watcom all do the decay on both sides. Borland does the decay on the argument side, but only when the template parameter is a pointer type. cfront does the decay on the argument side, but does not allow template parameters of array type.

To summarize, all of the compilers to which I have access do the decay on the argument side in some or all cases. Most also do the decay on the parameter side.

Answer:

Version added: 17
Version updated: 17

Member Template Issues

8.3 Question: Can a member function template be used as a copy constructor or copy assignment operator?

Status: Approved in Kona

```
struct A {
        A();
        template <class T> A(const T&);
        template <class T> operator =(const T&);
};

int main()
{
        A a1;
        A a2(a1);  // Implicitly generated copy or template?
        a1 = a2;   // Implicitly generated assignment or template?
}
```

Answer: No, a member function template cannot be used as a copy constructor or copy assignment operator. The copy constructor and copy assignment are special operations, and the existence of a template that could potentially generate such a function should not be taken to mean that the user wants the template version to be used in place of the implicitly generated function.

If the user wants the template to be used, an explicitly written function that calls the template version must be written.

If we were to decide that the templates could be used for this purpose, there would be no way for a user to request that the implicitly generated function should be used in place of the template.

Version added: 15
Version updated: 15

8.10 Question: What kind of entity can appear in a template friend declaration?

Status: Open

The purpose of this issue is to clarify the rules regarding the matching of a template friend declaration with a prior declaration of a template.

Answer: A template friend declaration that refers to a member function template must match the previous declaration of the template. It is not possible to have a "partial friend" declaration in which some of the template parameters are bound to specific types.

```
template <class T> struct A {
        template <class T2> void f(T2);
};

template <class U> class B {
        template <class T>
        template <class T2> friend void A<T>::f(T2); // okay

        template <class T>
        template <class T2> friend void A<T2>::f(T); // error

        template <>
        template <class T2> friend void A<U>::f(T2); // error

        template <>
        template <class T2> friend void A<int>::f(T2); // error
};
```

Version added: 17
Version updated: 17