X3J16/97-0087
WG21/N1125

Template Issues and Proposed Resolutions
Revision 21

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

September 30, 1997

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.
Version 18 (97-0015/N1053) – March 7, 1997: Distributed at the Nashua meeting and in the post-Nashua mailing. Contains additional new issues.
Version 19 (97-0045/N1083) – June 3, 1997: Distributed in the pre-London mailing. Reflects tentative decisions made in Nashua and at the template meeting in May plus additional new issues.
Version 20 (97-0077/N1115) – September 29, 1997: Distributed in the pre-Morristown mailing. Reflects decisions made at the London meeting.
Version 21 (97-0087/N1125) – September 30, 1997: Distributed in the pre-Morristown mailing. Contains the remaining open issues and new issues.

Summary of Issues

Other Issues

6.60 What are the semantics of a friend instance declaration?

6.61 Types vs. nontypes in template argument lists

6.62 Lookup of `f<...>` in friend declaration.

6.63 Lookup of `p->f<...`.

6.64 Clarification of explicit template argument list evaluation.

6.65 Clarification of function parameter type adjustments when using explicit function template arguments.

Member Template Issues

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

Other Issues

6.60 Question: What are the semantics of a friend instance declaration?

Status: Open

Answer:

A friend instance declaration is a friend declaration that refers to an instance (either generated or explicitly specialized) of a template.

When a friend declaration refers to an entity using an unqualified name, an explicit template argument list must be supplied to indicate that the declaration refers to the template and should not be considered to declare a new function.

```
template <class T> void f(T);

template <class T> struct A {
        friend void f<>(T);  // refers to template
        friend void f(T*);   // declares a new function
};
```

When the friend declaration refers to an entity using a qualified name, an explicit template argument list is only required if the overload set that is named contains both a template and nontemplate function that match the specified type (because a declaration that uses a qualified name always refers to a previously declared entity). For example,

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

class B {
        friend void A<int>::f(int);     // nontemplate
        friend void A<int>::f<>(int);   // template
        friend void A<int>::g(int);     // template
};
```

These rules are consistent with the rules for calling a function, in which overload resolution is used to select the best matching function, but an explicit template argument list, such as `<>`, can be used to exclude nontemplates from the set of functions that are considered.

A function cannot be defined in a friend instance declaration. An explicit specialization must be used for this purpose.

Default arguments may not be specified in a friend instance declaration. An instance does not participate in overload resolution, so such default arguments would be useless (which is why they are already prohibited in explicit specialization declarations).

The `inline` specifier may not be used in a friend instance declaration. Whether or not an instance is inline is determined by the template declaration, and any specialization declaration that may apply. It should not be possible to alter this in a friend declaration.

Version added: 19
Version updated: 19

6.61 Question: Types vs. nontypes in template argument lists

Status: Open

14.3 [temp.arg] paragraph 2 indicates that an ambiguity between a type-id and an expression is resolved to a type-id, but it does not make clear when such an ambiguity is considered to exist. For example, if you have a class template declaration that is known to take an integer argument, should `int()` be parsed as an expression?

```
template <class T> struct A {};
template <int I> struct B {};

template <class T> void f(T);
template <int I> void f(T);

A<int()> a;
B<int()> b;

template void f<int()>(int());
```

On the other hand, ambiguities can exist for both class and function template references, as illustrated by this example in which the class template is itself dependent on a template parameter.

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

Answer: The normal disambiguation process is always used to determine whether a given template argument is a type or nontype, even when the argument is used in a context in which the corresponding template parameter is known to be a type or nontype.

Version added: 19
Version updated: 19

6.62 Lookup of `f<...>` in friend declaration.

Status: Open

How is a name followed by an explicit template argument list looked up in a friend function declaration? For example:

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

The basic question is whether this is considered a reference to a previously declared `f`, or a declaration of a new `f`. The proposed answer is that this is a reference to a previously declared `f` and as such, that it is looked up using the normal lookup rules for unqualified name lookup specified in 3.4.1.

The semantics of a reference like this differ from the handling of unqualified friend declarations without an explicit argument list. A declaration such as "`friend void f(int)`" declares a member of the nearest enclosing namespace, and never refers to the a member of the current class or a function outside of the nearest enclosing namespace. Such differences are unfortunate, but I think that the differences are sensible and necessary.

These differences are necessary as a result of the fact that when you see a name followed by a “`<`” you need to look up the name to determine whether the `<` is a less than sign or the start of a template argument list.

This means that, for name lookup purposes, the example above is a reference to the name `f` while the example below is a declaration of `f` In other words, the example above is ill-formed if the template `f` was not previously declared, while the example below is valid whether or not the global scope includes a function or template named `f`.

```
struct A {
        void f(int);
        friend void f(int); // declares ::f
};
```

Similarly, a name declared in a friend declaration is a member of the nearest enclosing namespace and the search for a previous declaration extends only as far as that namespace. In the following example, there is no reason for the “`friend void f<int>(int)`” declaration not to find the template declared in namespace `N`. The “`friend void g(int)`” declaration, on the other hand, declares `O::g` because the search for a previous declaration does not extend to namespace `N`.

```
namespace N {
        template <class T> void f(T);
        void g(int);
        namespace O {
                struct A {
                        friend void f<int>(int); // N::f
                        friend void g(int); // declares O::g
                };
        }
}
```

The one unfortunate consequence of this rule is that special care needs to be taken when a friend declaration with an explicit template argument list refers to a name that is also a member of the current class. In such cases, a qualified name must be used in order to refer to the template from the outer scope.

```
template <class T> void f(T) {}
struct A {
        template <class T> void f(T) {}
        friend void ::f<int>(int);
};
```

Answer: An unqualified declarator with an explicit template argument list in a friend declaration is looked up using the normal lookup rules for an unqualified name.

Version added: 21
Version updated: 21

6.63 Lookup of `p->f<...`.

Status: Open

How is `f` looked up in an expression like `p->f<...`.

In the following example, it is clear that the `f` should refer to the template `A::f`.

```
struct A {
        template <class T> void f(T);
        void g(A* p) {
                p->f<int>(1);
        }
};
```

Similarly, I believe the following usage is currently permitted by the working paper. The working paper specifies special rules for `p->class-name-or-namespace-name::...`, and a `template-id` is a `class-name`, so presumably this should work.

```
template <class T> struct f {
        struct B {
                int i;
        };
};

struct A : public f<int>::B {
        void g(A* p) {
                p->f<int>::B::i = 1;
        }
};
```

In both of these examples the compiler sees `p->f<`, at which point it has to decide what to do with `f`. It is not reasonable, in my opinion, to attempt to scan forward and determine whether a `>::` exists that matches the `f<`.

The question that this is leading up to, of course, is how to handle examples like the following:

```
template <class T> struct f {
        struct B {
                int i;
        };
};

struct A : public f<int>::B {
        template <class T> void f(T);
        void g(A* p) {
                // Which of the following is permitted?
                p->f<int>(1);
                p->f<int>::B::i = 1;
        }
};
```

Answer:

The proposed resolution is: If the id-expression is of the form `p->identifier<...`, the identifier is first looked up in the class of the object expression. If the identifier is not found, it is then looked up in the context of the entire postfix-expression. The program is ill-formed if the name, when looked up in the context of the entire postfix expression, does not name a class or function template. If the lookup in the class of the object expression finds a template, the name is also looked up in the context of the entire postfix-expression. If the name is found in the context of the entire postfix-expression, and the name found is a template, it must refer to the same entity as the one found in the class of the object expression, otherwise the program is ill-formed.

This differs from the rule for looking up `A` in `p->A::B`, but a different rule is needed for this case to avoid breaking code. For example, the addition of the global template `f` would render the code ill-formed if the lookup used rules similar to the ones used for `p->A::B`.

```
template <int I> void f();
struct A {
        int f;
        void g(A* p) {
                bool b = p->f < 1;
        }
};
```

Version added: 21
Version updated: 21

6.64 Clarification of explicit template argument list evaluation.

Status: Open

The subclause that describes the explicit specification of function template arguments (14.8.1 [temp.arg.explicit]) does not fully describe the steps in which a particular function template specialization is selected when a set of overloaded function templates is involved.

For example, the WP does not specify the status of the following code. The significant issue in this example is that `int` is not a valid template argument for template #2 because "`int int::*` is not a valid type.

```
template <class T> int f(T*);       // #1
template <class T> int f(int T::*); // #2

struct A { int i; };

int i1 = f<int>(0);
```

There are at least two possible interpretations of this example:

1. The code is well formed, and template #1 is called because template #2 would not be valid.
2. The code is ill-formed because template #2 would not be valid.

What if the type of one of the template arguments does not match the parameter type?

```
template <class T, T> int f(int);
template <class T, T*> int f(int);

int i1 = f<int,0>(0);  // ambiguous
int i2 = f<int,1>(0);  // 1 can't be converted to int*
```

What if a nontype argument has a type that can be converted to the type of the corresponding parameter, but the value is out of range?

```
template <int> int f(int);
template <char> int f(int);

int i1 = f<1>(0);    // ambiguous
int i2 = f<1000>(0); // is this ambiguous?
```

Answer:

The proposed answer is that the attempt to create an invalid type while evaluating the template arguments or the function type result in a given function template from being eliminated from the set of templates that are considered for a given reference, but value-based errors do not do so.

Specifically, the following conditions disqualify a function template from being used:

1. Attempting to create an array with an element type that is `void`, a function type, or a reference type, for example:
   ```
   template <class T> int f(T[5]);
   int i = f<int>(0);
   int j = f<void>(0);
   ```
2. Attempting to use a type that is not a class type in a qualified name.
   ```
   template <class T> int f(typename T::B*);
   int i = f<int>(0);
   ```
3. Attempting to use a type in the qualifier portion of a qualified name that names a type when the type does not contain the specified type member.
   ```
   template <class T> int f(typename T::B*);
   struct A { };
   struct C { int B; };
   int i = f<A>(0);
   int j = f<C>(0);
   ```
4. Attempting to create a pointer to a reference type.
5. Attempting to create a reference to a reference type or to void.
6. Attempting to create a pointer to member with a type that is not a class type.
   ```
   template <class T> int f(int T::*); // #2
   int i = f<int>(0);
   ```
7. Attempting to perform an invalid conversion in either a template argument expression, or an expression used in the function declaration.
   ```
   template <class T, T*> int f(int);
   int i2 = f<int,1>(0);  // 1 can't be converted to int*
   ```

```
template <class T> int f(int[(T)1]);
int i = f<int*>(0);  // 1 cannot be converted to int*
```

Note that these conditions only disqualify a function template if they occur in the template parameter list or the function type. If, after selecting a given function template, such an error were to occur in an exception specification, the program would be ill-formed.

Version added: 21
Version updated: 21

6.65 Clarification of function parameter type adjustments when using explicit function template arguments.

Status: Open

There are several transformations that take place on the parameter types of functions. Top level qualifiers are removed, function types decay to pointers to functions, and array types decay to pointers.

When an explicitly specified template argument is used as a function parameter type, the parameter type undergoes the normal parameter transformations. Note that it is the function parameter type that is transformed, not the template argument itself.

```
template <class T> void f(T t) { t++; }
template <class X> void g(const X x) { x++; }
template <class Z> void h(Z, Z*);

int main()
{
        // #1: function type is f(int), t is nonconst
        f<int>(1);
        // #2: function type is f(int), t is const
        f<const int>(1);
        // #3: function type is g(int), x is const
        g<int>(1);
        // #4: function type is g(int), x is const
        g<const int>(1);
        // #5: function type is h(int, const int*)
        h<const int>(1,0);
}
```

Version added: 21
Version updated: 21

Member Template Issues

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

Status: Approved by core-3 in Nashua. Is this in the WP?

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