X3J16/94-0125
WG21/N0512

Template Issues and Proposed Resolutions
Revision 8

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

November 3, 1994

Introduction

This version contains only the issues resolved at the Waterloo meeting.

Class Template References

2.22 Proposed revision to rules for explicit instantiation of all class members.

In the current WP an explicit instantiation request of a template class implies the instantiation of all its members. I propose that this be revised to say that it implies the instantiation of all its members that have not been specialized.

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

template <class T> void A<T>::f(){}
template <class T> void A<T>::g(){}
void A<int>::h(){}

template A<int>;  // Instantiates A<int>::f and A<int>::g
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

Function Templates

3.15 Question: How may elaborated type specifiers be used in function template declarations?

```
template <class T> void f(struct T t){} // Error
template <class T> void f(union T t){}  // Error
```

```
template <class T> void f(enum T t){}   // Error

union U {};
struct S {};
class C {};
enum E {};

int main()
{
        U u;
        S s;
        C c;
        E e;

        f(u);
        f(s);
        f(c);
        f(e);
}
```

Answer: An elaborated type specifier containing a template parameter name may not be used in a function template declaration.

Status: Approved in Waterloo

Version added: 7
Version updated: 8

3.16 Clarification of template parameter deduction rules.

The WP does not currently describe how the type deduction process works when multiple function arguments are used to deduce a single type. I believe that there is general agreement on how this is done, but the WP needs to be explicit about this process.

Proposed clarification: Template parameters that are not explicitly specified must be deducible from the actual arguments of a given call (such parameters will be referred to as deducible parameters). A set of template parameter values (types and nontypes) is produced for each function parameter containing deducible parameters. Each function parameter is deduced independently of any other parameters (i.e., the deduction of one parameter does not bias the deduction of a subsequent parameter). The set of parameter values deduced from a function parameter must be consistent with the values deduced from previous parameters (i.e., one can determine that a given template fails to match a call when a parameter value deduced from one function parameter is inconsistant with the value deduced from a previous function parameter).

In the following example, both calls are ill-formed beacuse the values of `T` deduced for each of the function template's function parameters are not consistent with one another.

Some compilers incorrectly accept the first call while rejecting the second call. These compilers incorrectly perform a derived to base conversion on the second argument. In other words, the evaluation of the first function parameter biases the deduction of the second. The type deduction process should not exhibit this kind of order dependency.

```
template <class T> void f(T, T){}

struct A {};
struct B : public A {};

int main()
{
        A       a;
        B       b;

        f(a, b);  // Error - no matching function
        f(b, a);  // Error - no matching function
}
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

3.17 Question: How may an overloaded function name be used as a function template argument in a context that requires parameter deduction?

Answer: If the address of an overloaded function is used as an argument in a function template call, the compiler attempts to match each member of the set of overloaded functions with the function template parameter. The result must be a single nontemplate function or a template function reference in which all of the template parameters have been explicitly specified (i.e., in which no type deduction is required).

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

void g(int,int);
void g(char,int);

void h(char,int);
void h(int,int,int);

int main()
{
        f(g);  // Error - ambiguous
        f(h);  // OK - only h(char, int) matches
}
```

The following is another example using member pointers instead of normal pointers:

```
struct A {
        void f(int){}
        void f(int, int){}
};

template<class T1, class T2> void g(T1* t, void (T1::*func)(T2)){}
```

```
main() {
        A a;
        g(&a, &A::f);  // OK - only A::f(int) matches
}
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

3.18 Quesiton: Must a function template declaration be visible when an instance of the template is called?

file1.c:

```
template <class T> void f(T){}
int main()
{
        f(1);
        some_function();
}
```

file2.c:

```
void f(int);
void some_function()
{
        f(1);  // Error (although not a required diagnostic)
}
```

Answer: Yes. If the definition of a function is to be supplied by a generated compiler instance, the template declaration must be visible at the point of the call. If the definition is to be supplied by a user specialization, both the template declaration and the specialization declaration must be visible.

Note: A compiler could diagnose this kind of error by using a different name mangling scheme for template and nontemplate functions and detecting the presence of both template and nontemplate varieties of the same name.

Status: Approved in Waterloo

Version added: 7
Version updated: 7

3.19 What are the rules regarding the deduction of template template parameters?

Answer: A template template parameter may only be deduced from a template template parameter of a template class instance used in the argument list of the call.

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

```
int main()
{
        A<B> ab;
        f(ab);  // Calls f(A<B>)
}
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

Member Function Templates

4.7 Question: Can a member function of a class specialization be instantiated from a member function of the class template? (This is an issue raised by Erwin Unruh). I believe this is a clarification of existing practice.

Answer: No. In the example below, `A<int>::f()` is undefined and would result in a linker error. The same rule applies to static data members of class specializations.

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

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

int main()
{
        A<int> a;
        a.f();
}
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

4.8 Question: Can a template member function be declared in a specialization declaration?

Answer: Yes. (However, see also 6.18)

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

```
void A<int>::f();  // OK - A<int>::f will not be generated from
                   // the template

int main()
{
        A<int> a;
        a.f();
}
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

4.9 Question: Can a member function defined in a class template definition be specialized?

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

template <class T> void A<T>::g(){}

void A<int>::f(){}  // OK
void A<int>::g(){}  // OK
```

Answer: Yes

Status: Approved in Waterloo

Version added: 7
Version updated: 8

Other Issues

6.17 Question: Can pointer to member types be used as nontype parameters?

Answer: Yes. The actual argument may be a pointer to a member of the specified class or of a class derived from the specified class.

```
struct A {
        int i;
        void f();
};
struct A2 : public A {};

template <int A::* pma> struct B {};
template <void (A::* pmfa)()> struct C {};

B<&A::i> b1;
C<&A::f> c1;
B<&A2::i> b2;
C<&A2::f> c2;
```

Status: Approved in Waterloo

Version added: 7
Version updated: 7

6.19 Clarification of explicit designation of a name as a type.

The WP (14.2) says that in an explicit type designation such as

`typedef` *qualified-name*;

the leftmost identifier of the *qualified-name* must be a *template-argument* name. This needs to be revised because the type designations are also needed for members of base classes whose type depends on a template parameter.

This should be revised to say that the *qualified-name* must include a qualifier containing a template parameter or template class name.

Status: Approved in Waterloo

Version added: 7
Version updated: 7