X3J16/96-0153
WG21/N0971
July 9th, 1996
John H. Spicer, Edison Design Group

Template Issue Resolutions from the Stockholm Meeting

This document shows the proposed WP changes for resolutions to issues raised in 96-0094/N0912 “Template Issues and Proposed Resolutions - Revision 15”.

1. Working paper changes for issue 3.30:

Add the following before box 31 in 14.8.2 [temp.deduct]:

> In a type of the form T::X (e.g., A<T>::X, A<i>::X, etc.), template argument values cannot be deduced from the qualifier. When a template parameter is used in this context, an argument value that has been explicitly specified, or deduced from other arguments is used. If the value cannot be deduced elsewhere, and is not explicitly specified, the program is ill-formed. Implicit conversions (4) will be performed on a function argument that corresponds with a function parameter that contains only non-deducible template parameters and explicitly specified template parameters (14.8.1 [temp.arg.explicit]).

2. Working paper changes for issue 5.5:

Add the following before 14.7.3 [templ.expl.spec] before paragraph 2:

> An explicit specialization must be declared in the namespace of which it is a member, or, for class members, in the namespace of which the class is a member. Such a declaration may also be a definition. If the declaration is not a definition, the specializarion may be defined later in the namespace in which it was declared, or in a namespace that encloses the one in which it was declared.

3. Working paper changes for issue 5.7:

Replace the last two sentences of 14.7.3p15 [templ.expl.spec] (Thus, an explicit ... in some translation unit) with:

> 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 the 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.

4. Working paper changes for issue 5.8:

Add to the end of section 14.7.3 [templ.expl.spec]:

> An explicit specialization declaration shall not be a friend declaration.

5. Working paper changes for issue 5.9 and 6.44:

Replace the first sentence of 15.4 [except.spec] paragraph 2 with:

> If any declaration of a function has an exception-specification, all declarations, including the definition of that function and an explicit specialization of that function shall have an exception-specification with the same set of type-ids. An exception-specification may be specified, but is not required to be specified, in an explicit instantiation directive. If an exception-specification is specified, it shall have the same set of type-ids as other declarations of that function.

6. Working paper changes for issue 5.10:

Add after sentence of 7.1.1p5 [dcl.stc]:

> A storage class shall not be specified in an explicit specialization or an explicit instantiation directive.

7. Working paper changes for issue 6.42:

Add to the end of 3.7.3.1p1:

> A function template can be an allocation function. Such a template shall declare its return type and first parameter as specified above (i.e., template parameter types shall not be used in the return type and first parameter type). Template allocation functions shall have two or more parameters.

Add to the end of 3.7.3.2 [basic.stc.dynamic.deallocation]:

> A function template can be an deallocation function. Such a template shall declare its return type and first parameter as specified above (i.e., template parameter types shall not be used in the return type and first parameter type). Template deallocation functions shall have two or more parameters. However, a template deallocation function will never be used to generate the two parameter version of a member deallocation function (i.e., the one whose second parameter is of type size_t).

8. Working paper changes for issue 6.43:

Add after 14 [temp] paragraph 6:

> A template-declaration, explicit specialization, or explicit instantiation directive shall contain at most one declarator. When such a declaration is used to declare a class, no declarator is permitted.

9. Working paper changes for 6.45:

Add to the end of 14.5.5 [temp.fct]:

> A function template can be overloaded with other function templates and with normal (non-template) functions. A normal function is never related to a function template, even if it has the same name and type as a potentially generated function template specialization. Footnote: That is, guiding declarations have been removed from the language.

Remove 14.8.4 [temp.over.spec].

Remove 14.8.3 [temp.over] paragraph 3.

10. Working paper changes for 6.47:

Add to 14.5.3 [temp.friend] following paragraph 2:

> When a function is defined in a friend function declaration in a class template, the function is defined when the class is specialized. The function is defined even if it is never used.

11. Working paper changes for 6.48:

Add to the end of 14.5.3 [temp.friend]:

> A friend template shall not be declared in a local class.

12. Working paper changes 7.10:

In 14.3p3 replace "integral type, the address of an object or" with "integral type, the name of a non-type non-reference template-parameter, the address of an object or".

In 5.19p1, replace "and sizeof expressions" with "non-type template parameters of integral or enumeration types, and sizeof expressions".

In 5.19p4, replace "implicitly using" with "implicitly using a non-type template parameter of pointer type, or implicitly using".

In 5.19p5, replace "or a function" with ", a non-type template parameter of reference type, or a function".

13. Working paper changes for 8.1, 8.2, 8.5, and 8.7

Remove 14.5.2 [temp.mem] paragraph 2.

Add to the end of 14.5.2 [temp.mem]:

> A local class shall not have member templates. Access control rules apply to member template names. A normal (non-template) member function with a given name and type and a member function template of the same name that could be used to generate a specialization of the same type can be both be declared in a class. When both exist, a reference refers to the non-template unless an explicit template argument list is supplied.
>
> [Example:

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

template <> void A<int>::f(int) {}  // non-template member
template <> template <> void A<int>::f<>(int) {}  // template member

int main()
```

```
{
      A<char> ac;
      ac.f(1);  // non-template
      ac.f('c');  // template
      ac.f<>(1);  // template
}
]
```

A member function template shall not be virtual. A specialization of a member function template does not override a virtual function from a base class.

[Example:
```
class B {
      virtual void f(int);
};

class D : public B {
      template <class T> void f(T);  // does not override
                                     // B::f(int)
      void f(int i) { f<>(i); }  // overriding function
                                 // that calls template
};
```
]

A specialization of a template conversion operator is referenced in the same way as a non-template conversion operator that converts to the same type.

[Example:
```
struct A {
      template <class T> operator T*();
};
template <class T> A::operator T*(){ return 0; }
template <> A::operator char*(){ return 0; }  // specialization
template A::operator void*();        // instantiation

int main()
{
      A      a;
      int*   ip;

      ip = a.operator int*();  // explicit call
}
```
]

If more than one conversion template can produce the required type, the partial ordering rules [temp.func.order] are used to select the ``most specialized'' version of the template that can produce the required type. Note that, as with other conversion functions, the type of the implicit this parameter is not considered (i.e., members of base classes are considered equally with members of the derived class, except that a derived class conversion function hides a base class conversion function that converts to the same type).

14. Working paper changes for issue 8.4:

In 13.1 [over.load], paragraph 2, in the second bullet item, following the first sentence, add the following:

> Likewise, member function template declarations with the same name, the same parameter types, and the same template parameters cannot be overloaded if any of them is a static member function template declaration.

15. Working paper changes for issue 8.6:

Add to the end of 14.8.1 [temp.arg.explicit]:

> Because the explicit template argument list follows the function template name, and because conversion member function templates and constructor member function templates are called without using the functions name, there is no way to provide an explicit template argument list for these function templates.