X3J16/96-0094
WG21/N0912

Template Issues and Proposed Resolutions

Revision 15

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

May 28, 1996

Revision History

Version 1 (93-0039/N0246) – March 5, 1993: Distributed in Portland and in the post-Portland mailing.
Version 2 (93-0074/N0281) – May 28, 1993: Distributed in pre-Munich mailing. Reflects tentative decisions made in Portland and additional issues added after the Portland meeting. In Portland, the extensions working group reviewed most of the issues from 1.1 to 2.8 and also reviewed 6.3.
Version 3 (93-0123/N0330) – September 28, 1993: Distributed in pre-San Jose mailing. Reflects decisions made in Munich. No new issues were added in this revision.
Version 4 (93-0183/N0330) – November 24, 1993: Distributed in post-San Jose mailing. Reflects decisions made in San Jose. Note that issues that have been closed as a result of formal motions in San Jose will be omitted from subsequent versions of this paper. In San Jose the extensions working group identified a number of issues that required additional work. These issues have not been addressed in this paper but will be addressed in the next revision.
Version 5 (94-0020/N0407) – January 25, 1994: Distributed in the Pre-San Diego mailing. The 41 closed issues have been removed, 20 have been added, and a few existing ones have been updated.
Version 6 (94-0068/N0455) – March 25, 1994: Distributed in the Post-San Diego mailing. Reflects decisions made in San Diego. Note that issues that have been closed as a result of formal motions in San Diego will be omitted from subsequent versions of this paper. In San Diego the extensions working group identified a number of issues that required additional work. These issues have not been addressed in this paper but will be addressed in the next revision.
Version 7 (94-0096/N0483) – June 1, 1994: Distributed in the Pre-Waterloo mailing. The 24 issues closed in version 6 have been removed and 16 new issues have been added.
Version 8 (94-0125/N0512) – November 3, 1994: Distributed in Valley Forge and in the post-Valley Forge mailing. Reflects decisions made in Waterloo. This version contains only issues closed in Waterloo. Version 9 will be distributed at the same time as version 8 and will contain the open issues and new issues.
Version 9 (94-0200/N0587) – November 5, 1994: Distributed in Valley Forge and in the post-Valley Forge mailing. Issues closed in version 8 have been removed and new issues have been added.
Version 10 (94-0212/N0599) – November 25, 1994: Distributed in the post-Valley Forge mailing. Reflects decisions made in Valley Forge. Includes a number of new issues supplied by

Erwin Unruh.
Version 11 (95-0007/N0607) – January 31, 1995: Distributed in the pre-Austin mailing. Includes a few new issues.
Version 12 (95-0101/N0701) – May 28, 1995: Distributed in the pre-Monterey mailing. Reflects decisions made in Austin. 9 issues have been closed, 12 new issues have been added.
Version 13 (95-0158/N0758) – July 20, 1995: Distributed in the post-Monterey mailing. Reflects decisions made in Monterey.
Version 14 (96-0023/N0841) – January 30, 1996: Distributed in the pre-Santa Cruz mailing.
Version 15 (96-0094/N0912) – May 28, 1996: Distributed in the pre-Stockholm mailing. Reflects decisions made in Santa Cruz and contains new issues.

Introduction

This document attempts to clarify a number of template issues that are currently either undefined or incompletely specified. In general, this document addresses smaller issues.

Of the issues that are addressed, some are covered in far more detail than others. Some of the resolutions represent solid proposals while others are more like trial balloons. The more tentative proposals are so designated in the body of the document.

Even those resolutions that represent fairly solid proposals are *only* proposals. This document is not intended as a formal proposal of any specific language changes. Rather, it is intended as to be used as a framework for discussion of these issues. Hopefully this will ultimately result in formal proposals for language changes.

Organization of the Document

The document is organized in sections. Each section consists of a list of questions. Each question has an answer, a status, the version number of the first version of this document that included the question, and the version number of the last change in the question. This allows the reader to skip over questions that have not changed since the last time he or she read the document.

Acknowledgements

I would like to thank Bjarne Stroustrup who contributed greatly by providing issues, reviewing and improving upon proposed resolutions, and providing insights into other language changes that may impact templates. Thank you to Erwin Unruh, who has contributed to many of the issues, and who also contributed the "Erwin Unruh's Issues" section. Thank you to Mike Karasick and Lee Nackman (and possibly others) from IBM who contributed issues concerning name binding and member functions of partial specializations of class templates.

Summary of Issues

Because this is a rather long document this summary is provided to allow the reader to quickly find issues in which he or she may be interested. Note that closed issues have been removed from the body of the paper. Please refer to a previous version of the paper for additional information on these issues.

Template Parameters

1.1 Can template parameters have default arguments? (closed in version 4)

1.2 Where can default arguments for template parameters be specified? (closed in version 4)

1.3 Can a type parameter be used in the type declaration of a nontype parameter? (closed in version 4)

1.4 Can a nontype parameter as used above have a default argument? (closed in version 4)

1.5 Should it be possible to redeclare a template parameter name to mean something else inside a template definition? (closed in version 4)

1.6 Can the name of a nontype parameter be omitted? (closed in version 4)

1.7 Can the name of a type parameter be omitted? (closed in version 4)

1.8 Can a typedef appear in a template declaration? (closed in version 4)

1.9 Can a nontype parameter have a reference type? (closed in version 4)

1.10 Are qualifiers allowed on nontype parameters? (closed in version 4)

1.11 May a template parameter have the same name as the class template with which it is associated? (closed in version 4)

Class Template References

2.1 Can a nontype parameter that is not a reference be used as an lvalue or have its address taken? (closed in version 4)

2.2 Can the class template name be used as a synonym for the current instantiation inside a class template? (closed in version 4)

2.3 Can a class template have a template parameter as a base class? (closed in version 4)

2.4 Can a local type be used as a type argument of a class template? (closed in version 4)

2.5 Can a character string be a nontype argument? (closed in version 4)

2.6 Can any conversions be done on nontype actual arguments of class templates? (closed in version 6)

2.7 What causes a template class to be instantiated? (closed in version 4)

2.8 How can a class template name be used within the definition of the template? (closed in version 6)

2.9 The previous rule makes possible runaway recursive instantiations. How should an implementation prevent this? (closed in version 5)

2.10 At what point are names injected? (closed in version 6)

2.11 Does an array parameter decay to a pointer type? (closed in version 6)

2.12 What can be used as an actual argument for a parameter that is a reference? (closed in version 4)

2.13 Can template parameters be used in elaborated type specifiers? (closed in version 4)

2.14 Can a class template or function template be declared as a friend of a class? (closed in version 6)

2.15 Can template arguments be supplied in explicit destructor calls? (closed in version 4)

2.16 What happens if the same name is used for a template parameter of an out-of-class definition of a member of a class template and a member of the class? (closed in version 6)

2.17 What happens if the name of a template parameter of a class template is also the name of a member of one of its base classes? (closed in version 6)

2.18 When must a type used within a template be completed? (closed in version 6)

2.19 Must a specialization declaration precede the use of a class template in a context that requires only an incomplete type? (closed in version 6)

2.20 Proposal to defer error checking for `operator ->`. (closed in version 6)

2.21 When are names considered known in a template dependent base class? (closed in version 6)

2.22 Proposed revision to rules for explicit instantiation of all class members. (closed in version 8)

2.23 How does name injection interact with the semantics of friend declarations? (withdrawn - last in version 10)

2.24 Class template partial specialization clarification. (closed in version 13)

2.25 May a nested class within a class template be defined outside of the template? (closed in version 13)

2.26 Question: May a class nested within a template be declared as a template friend? (closed in version 13)

2.27 May a friend function be defined in a template friend declaration? (closed in version 13)

2.28 Clarification of specialization rules for nested classes. (closed in version 15)

2.29 Can a non-autonomous nested class be specialized? (closed in version 15)

2.30 Can nested classes and member template classes be specialized? (closed in version 15)

Function Templates

3.1 Can function templates have default function parameters? (closed in version 4)

3.2 Can the parameters with default arguments involve template parameters in their types? (closed in version 5)

3.3 Can a local type be used as a type argument of a template function? (closed in version 4)

3.4 Can any conversions be done when matching arguments to function templates? (closed in version 5)

3.5 The WP requires that every template parameter be used in an argument type of a function template. What constitutes a "use" of a template parameter in an argument type? (closed in version 4)

3.6 Can unnamed types be used as template arguments? (closed in version 4)

3.7 Can template parameters be used in qualified names in function template declarations? (closed in version 12)

3.8 Can a noninline function template be instantiated when referenced? (closed in version 4)

3.9 A proposal to allow conversions in function template calls. (closed in version 6)

3.10 What happens when the explicit specification of function template arguments results in an invalid type? (closed in version 6)

3.11 How do default arguments work when using new explicit specialization declarations? (closed in version 6)

3.12 How do old style specialization declarations interact with new style ones? (closed in version 6)

3.13 Revisiting default arguments. (closed in version 12)

3.14 What are the rules regarding use of the inline keyword in function template declarations? (closed in version 10)

3.15 How may elaborated type specifiers be used in function template declarations? (closed in version 8)

3.16 Clarification of template parameter deduction rules. (closed in version 8)

3.17 How may an overloaded function name be used as a function template argument in a context that requires parameter deduction? (closed in version 8)

3.18 Must a function template declaration be visible when an instance of the template is called? (closed in version 8) item[3.19] What are the rules regarding the deduction of template template parameters? (closed in version 8)

3.20 How are type/expression ambiguities resolved in explicitly qualified function template calls? (closed in version 10)

3.21 May template functions with the same signature coexist with one another? May a template function with a given signature coexist with a nontemplate function with the same signature. (closed in version 12)

3.22 Proposed rules for selecting between overloaded function templates (closed in version 12)

3.23 Binding of function and array types to template dependent reference parameters. (closed in version 15)

3.24 Clarification regarding nontype parameters deduced from array bounds. (closed in version 13)

3.25 Can a type parameter be deduced from the type of a nontype parameter? (closed in version 13)

3.26 What is the type of a constant deduced from an array bound? (closed in version 13)

3.27 Clarification of rules regarding expressions used as nontype arguments. (closed in version 13)

3.28 Elaborated type specifiers in function template declarations revisited. (closed in version 15)

3.29 Template argument deduction revisited. (closed in version 15)

3.30 How are nondeduced nested class references handled in function template declarations?

Member Function Templates

4.1 Are inline member functions that are not used by a given class template instance instantiated? (closed in version 4)

4.2 Can a noninline member function or a static data member be instantiated when referenced? (closed in version 4)

4.3 Must the template parameter names in a member function definition match the names used in the class definition? (closed in version 4)

4.4 What are the rules regarding use of the inline keyword in member function declarations? (closed in version 6)

4.5 How are default arguments for parameters of member functions of class templates handled? (closed in version 4)

4.6 Can a class template member function be redeclared outside of the class? (closed in version 6)

4.7 Can a member function of a class specialization be instantiated from a member function of the class template? (closed in version 8)

4.8 Can a template member function be declared in a specialization declaration? (closed in version 8)

4.9 Can a member function defined in a class template definition be specialized? (closed in version 8)

4.10 How are members of class templates declared and defined? (closed in version 13)

4.11 How are members functions of a partial specialization of a class template defined? (closed in version 13)

Explicit Specialization Issues

5.1 Can you create a specific definition of a class template for which only a declaration has been seen? (closed in version 4)

5.2 Can you declare an incompletely defined object type that is a specific definition of a class template? (closed in version 4)

5.3 Can the class template name be used as a synonym for the current specific definition inside the specific definition? (closed in version 4)

5.4 Can a specific definition of a class template be a local class? (closed in version 4)

5.5 Where can an explicit specialization be declared?

5.6 Clarification of rules regarding the explicit specialization of class templates.

5.7 How are the members of an explicitly specialized class defined?

5.8 What syntax is used to declare a template entity to be a friend?

5.9 What are the rules for exception specifications on explicit specializations?

5.10 What is the linkage (internal vs. external) of an explicit specialization?

Other Issues

6.1 Should classes used as template arguments have external linkage? (closed in version 4)

6.2 When must errors in template definitions be issued and when must they not be issued? (closed in version 4)

6.3 What kinds of types may be used in a function template declaration while still being able to deduce the template argument types? (closed in version 4)

6.4 Can a static data member of a class template be declared with an incomplete array type? (closed in version 4)

6.5 How should template arguments that contain "`>`" be parsed? (closed in version 4)

6.6 Can template versions of `operator new` and `operator delete` be declared? (closed in version 4)

6.7 How can a name that is undefined at the point of its use in a template declaration be determined to be a type or nontype? (closed in version 4)

6.8 May template declarations be given a linkage specification other than C++. (closed in version 6)

6.9 Should there be a translation limit that specifies a minimum depth of recursive instantiation that must be supported? (closed in version 6)

6.10 Can a single template declaration declare more than one thing? (closed in version 6)

6.11 Can a storage class be specified in a template parameter declaration? (closed in version 6)

6.12 Can an incomplete type be used as a template argument? (closed in version 6)

6.13 Can a template nontype parameter have a void type? (closed in version 6)

6.14 Can a nontype parameter be a floating point type? (closed in version 6)

6.15 What kind of expressions may be used as nontype template arguments?

6.16 Can a template parameter be used in an explicit destructor call? (closed in version 6)

6.17 Can pointer to member types be used as nontype parameters? (closed in version 8)

6.18 Issues regarding declarations of specializations. (closed in version 12)

6.19 Clarification of explicit designation of a name as a type. (closed in version 8)

6.20 Template compilation model proposal. (withdrawn - last in version 7)

6.21 How is a dependent name known to be a template? (closed in version 12)

6.22 Interaction of templates and namespaces. (closed in version 10)

6.23 Floating point template parameters revisited. (closed in version 10)

6.24 May function types be used as template parameters? (closed in version 12)

6.25 WP clarification: overloaded functions as template arguments (closed in version 10)

6.26 WP clarification: access checking an template arguments (closed in version 10)

6.27 Name binding problems (closed in version 12)

6.28 Can a user-specialization be provided for an `operator ->` that cannot be instantiated? (closed in version 13)

6.29 How are names from template dependent base classes to be used? (withdrawn, last in version 12)

6.30 When is a template argument list required in a function declaration? (closed in version 15)

6.31 Is a template argument list permitted in a function template declaration? (closed in version 15)

6.32 Can compiler-generated functions be explicitly specialized or instantiated? (closed in version 15)

6.33 When is a nested-name-specifier allowed in the declarator in an explicit instantiation. (closed in version 15)

6.34 Can an explicit instantiation that refers to a class be used to instantiate all the members of a nested class? (closed in version 15)

6.35 `typename` syntax problems. (closed in version 15)

6.36 Where is `typename` permitted? (closed in version 15)

6.37 Does `typename` affect name lookup? (closed in version 15)

6.38 Clarification of interaction of namespaces and specialization (closed in version 15)

6.39 Correction of default template argument description. (closed in version 15)

6.40 Clarification of access checkin in explicit instantiation directives. (closed in version 15)

6.41 Linkage consistency rules for specialization and guiding declarations. (closed in version 15)

6.42 Clarification of rules for template operator new and delete.

6.43 Clarification of rules for the number of things declared in a template declaration.

6.44 What are the rules for exception specifications on explicit instantiations?

6.45 A proposal to eliminate guiding declarations.

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?

6.48 Are template friend declarations permitted in local classes?

Erwin Unruh's Issues

7.1 Type deduction for conversion operators (closed in version 12)

7.2 How does type deduction interact with overloading (closed in version 13)

7.3 How does type deduction interact with conversions (removed in version 15)

7.4 What is the point of instantiation really? (closed in version 15)

7.5 Short addition to 3.17 (closed in version 13)

7.6 Type deduction with several results (closed in version 13)

Member Template Issues

8.1 Can normal members coexist with member function templates that could generate the same signature?

8.2 Clarification of rules for member templates and virtual functions.

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

8.4 Can two member templates coexist whose only difference is that one is static and the other is not? How are template conversion functions explicitly called, explicitly specialized, and explicitly instantiated.

8.6 Can a template argument list be supplied to an constructor template or conversion template?

8.7 How is a conversion function chosen when the set of conversions includes conversion templates functions?

8.8 Clarification of rules for standard conversions following template conversion functions

8.9 Can a member class template be declared and then defined later within the class?

Nontype Parameters for Function Templates

A proposal for nontype parameters for function templates as required by the `Bitset` class. (closed in version 4)

Class Template References

2.28 Clarification of specialization rules for nested classes.

Status: Approved in Santa Cruz.

The current wording in 7.1.5.3 [dcl.type.elab] does not permit an elaborated type specifier containing a qualified name to be the sole constituent of a declaration. Unless this is changed, it will not be possible to name a nested class member of a template class in an explicit instantiation.

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

template <> struct A<int>;     // okay
template <> struct A<char>::B; // not allowed by 7.1.5.3
```

Answer: 7.1.5.3 should be changed to permit this usage in explicit instantiations and explicit specializations.

Version added: 14
Version updated: 14

2.29 Question: Can a non-autonomous nested class be specialized?

Status: Approved in Santa Cruz

It is possible for the definition of a nested class to also be used to declare members of that class type. While it would still be possible to permit classes defined in such "non-autonomous" declarations to be specialized, it seems like a bad idea. Moreover, most such uses would require the nested class to be instantiated as part of the instantiation of the enclosing class anyway.

Answer: Yes, non-autonomous classes can be specialized (provided their use within the class does not force their instantiation, but this is a consequence of the normal rules that require a specialization to be declared before its first use).

```
template <class T> struct A {
        struct B {int i;} b;  // instantiated as part of A<T>
        union { int i; float f; };  // instantiated as part of A<T>
        struct C { long l; };  // not instantiated as part of A<T>
                               // so can be specialized
        struct D {int i;} *d;  // not instantiated as part of A<T>
                               // so can be specialized
};
```

Version added: 14
Version updated: 15

2.30 Question: Can nested classes and member template classes be specialized?

Status: Approved in Santa Cruz

In Austin, we disallowed specialization of member classes and member template classes. This was done because the rules that were then in effect concerning when nested classes were instantiated made such specializations impossible. In Monterey, we changed the rules making such specializations once again possible.

Answer: For consistency with other kinds of members, it is proposed that the ability to specialize member classes and member class templates be restored.

Version added: 14
Version updated: 14

Function Templates

3.23 Binding of function and array types to template dependent reference parameters.

Status: Rejected in Santa Cruz

WP 14.10.2 [temp.deduct] says that array and function types do not decay when binding to a parameter that is a reference. The problem with this is it permits array types to be used in places where the template writer had not intended them to be used. For example, the HP STL distribution includes a `max` template that is defined as:

```
template <class T>
inline const T& max(const T& a, const T& b) {
        return  a < b ? b : a;
}
```

This works well for most types, but fails for array types such as string literals.

```
int main()
{
        char* x;
        x = max("hello", "there");  // T is char[6]
        x = max("hi", "there");     // fails because T is char[3]
                                    // and char[6]
}
```

What is intended is that the resulting function parameter type for `const T&` is `const char*&`. What happens with the current WP wording is that the resulting function parameter is `const char (&)[6]`. This causes a problem: the length of the two strings must be identical for type deduction to succeed, and the return type will end up being a reference to array of the same size.

Answer: The proposed solution is to revise the deduction rules to say that an array or function type can only bind to a parameter that is declared with a reference to array or function type, as in the example that appears below.

More specifically, assuming `P` is the parameter type and `A` is the argument type: If `P` is a reference to an array type and `A` is an array type, or `P` is a reference to function type and `A` is a function type, and if the values of the all of the template parameters referenced by `P` can be deduced from `A`, then the original type of `A` is used for type deduction. Otherwise,

- if `A` is an array type, the result of the array to pointer decay is used in place of `A` for type deduction; otherwise,

- if `A` is a function type, the result of the function to pointer decay is used in place of `A` for type deduction.

```
template <class T, int I1, int I2>
T* f(T (&t1)[I1], T (&t2)[I2]);

int main()
{
        char* x;
        x = f("hello", "there");
}
```

This still permits binding of array types, but only in cases where that is explicitly indicated by the template writer.

Note that this illustrates another clarification that needs to be made. Major array bounds are part of the parameter type when the parameter is a reference. Consequently, nontype template parameters may be deduced from a major array bound in such cases.

Version added: 12
Version updated: 12

3.28 Elaborated type specifiers in function template declarations revisited.

Status: Approved in Santa Cruz

In Waterloo, we decided that an elaborated type specifier containing a template parameter name could not be used in a function template declaration.

Now that we have the partial ordering rules for function templates, this issue should be checked to see if it is still what we want.

With the partial ordering rules, we can now select one template over another based on one being "more specialized" than another. It seems that these rules could be applied to elaborated type specifiers as well.

If this is permitted in the partial ordering of function templates, it should also be permitted in the partial ordering used for class template partial specializations.

```
template <class T> class List {};
template <class T> void f(List<struct T> l){}  // #1
template <class T> void f(List<union T> l){}   // #2
template <class T> void f(List<enum T> l){}    // #3
template <class T> void f(List<T> l){}    // #3

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

int main()
{
        List<U> u;
```

```
        List<S> s;
        List<C> c;
        List<E> e;
        List<int> i;

        f(u);   // calls #2
        f(s);   // calls #1
        f(c);   // calls #1
        f(e);   // calls #3
        f(i);   // calls #4
}
```

Answer: Core-3 decided that the current rule banning use of template parameters in elaborated type specifiers only in function template declarations was not sufficient (because of things like partial specializations of classes), and that a simple prohibition against the use of template parameters in elaborated type specifiers was more desirable than a more complicated rule.

Consequently, template parameters cannot be used in elaborated type specifiers.

Version added: 14
Version updated: 15

3.29 Template argument deduction revisited.

Status: Part 1 approved in Santa Cruz, parts 2 and 3 rejected in Santa Cruz.

In Tokyo a number of template argument deduction cases were discussed. As a result, I was asked to reopen the issue of template argument deduction so that the following cases could be be reexamined:

```
template <template <class T1> struct X, class T2> void f(X<T2>); // #1
template <class T> void f(A<T>::B);  // #2
template <class T> void f(T::B);     // #3
```

1. It was pointed out that it is not currently possible to deduce a template template parameter from an actual argument whose type is a template instance, that this kind of deduction can readily be done, and that doing so provides significant functionality. For example, it permits writing of a function that operates on any of a number of different containers. For example,

```
template <class T struct List {};
template <class T struct Vector {};

template <template <class T1> struct Container, class Type>
void print(Container<Type>);
```

2. The second case is whether a template argument can be deduced from the parent class of a nested class or nested enumeration. This case is important to maintain the general rule that a nontemplate class can be converted to a template class. Without this deduction, nested classes within templates are severely limited. Furthermore,

without this rule member template classes are even more limited. The following example illustrates the kind of usage that is common for normal nested classes that cannot currently be done with nested classes and member templates of class templates.

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

template <class T> A<T>::B operator+(A<T>::B, A<T>::B);
template <class T1> template <class T2>
A<T1>::B<T2> operator+(A<T1>::B<T2>, A<T2>::B<T2>);
```

A member typedef is just a synonym for another type and so, of course, there is no way that the class containing the typedef can be deduced from an actual argument whose type was specified using the typedef.

3. The third case is a generalization of the second. This has been separated out because it was pointed out that some of the original objections to this issue when it was previously raised were primarily based on this more general form, which actually provides very little additional functionality over the more restricted version in #2.

Version added: 14
Version updated: 15

3.30 Question: How are nondeduced nested class references handled in function template declarations?

Status: Open

Core-3 has confirmed that a template parameter cannot be deduced from contexts such as the ones shown in the following example:

```
template <class T> void f(T::X);
template <class T> void f(A<T>::X);
```

But what about the usage like this:

```
template <class T> void f(T, T::X);
template <class T> void f(T, A<T>::X);
```

We know that `T` cannot be deduced from the second parameter of each function, but can the second parameter use the `T` deduced elsewhere in the parameter list? The alternatives are to make such usage ill-formed, or to specify that when a template parameter is used in a context in which its value cannot be deduced, the values deduced from elsewhere in the declaration are used. Relying on the values deduced elsewhere is consistent with the current handling of template parameters in function return types and exception specifications.

Angelika Langer, of Rogue Wave has a proposal in the pre-Stockholm mailing to simplify the interface of some of the STL routines using iterator traits. Her proposal is that interfaces of the form

```
template <class InputIterator, class T>
InputIterator find(InputIterator first,
                   InputIterator last,
                   const T&      value);
```

be replaced with

```
template <class InputIterator>
InputIterator find(InputIterator first,
                   InputIterator last,
    const typename iterator_traits<InputIterator>::value_type& value);
```

Angelika points out that the new interface is less error prone (because of the ability to ensure that the value type matches the iterator being used), results in the generation of fewer template instances (because of the ability to perform conversions on the arguments associated with nondeduced parameters), and simplicity.

Answer: When a template parameter is used in a type in a context in which its value cannot be deduced, and the value has not been explicitly specified, the value deduced elsewhere in the declaration is used. If the value cannot be deduced elsewhere and is not explicitly specified, the program is ill-formed. A function parameter containing only nondeducible parameter types is considered a nondedicible parameter for overload resolution purposes, meaning that the full set of conversions can be performed on arguments passed to that parameter.

Version added: 15
Version updated: 15

Explicit Specialization Issues

5.5 Question: Where can an explicit specialization be declared?

Status: Open

```
namespace N {
    class A {
        template <class T> void f(T);
        template <> void f(int){} // error -- specialization not
                                  // allowed in class scope
    };
    template <> void A::f(short);  // okay
    template <> void A::f(short){} // okay
    template <> void A::f(char);   // okay
}

template <>
void N::A::f(float);  // error -- specialization cannot be
                      // declared outside of its namespace

template <>
void N::A::f(char);   // error -- specialization cannot be
                      // redeclared outside of its namespace

template <> void N::A::f(char){}  // okay
```

Answer: An explicit specialization must be declared in the namespace in which the template was declared. If the template is a member template, the specialization must be declared in the namespace containing the enclosing class (see also issue 6.38). An explicit specialization may be defined (but not redeclared) later outside of the namespace of which it is a member.

Note: The difference between this and issue 6.38 is that this clarifies that an explicit specialization is not permitted in a class scope, and that an explicit specialization cannot be redeclared in an enclosing namespace scope.

Version added: 15
Version updated: 15

5.6 Clarification of rules regarding the explicit specialization of class templates.

Status: Open

The explicit specialization of a template must be declared before the first "use" of a template. What constitutes the "use" of a class template?

```
template <class T> struct A {
        template <class T2> struct B {};
};

A<char>::B<int> acbi;  // Generated from template

A<int>::B<int>* aibi;  // Complete instantiation not yet done

template <>
struct A<int>::B<char> {};  // Explicit specialization
                            // of instance of A<int>::B
A<int>::B<char> aibc;  // Uses specialization declared above

template <> template <class T>
struct A<int>::B {};  // Explicit specialization of template
```

Answer: A class template is "used" when an instance of the class is generated from the template. Consequently, a use of the template in such a way that does not require a full instantiation, or the explicit specialization of an instance of the template may precede the declaration of an explicit specialization of the template.

Version added: 15
Version updated: 15

5.7 Question: How are the members of an explicitly specialized class defined?

Status: Open

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

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

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

template <class T2> void A<int>::f(T2){}
void A<int>::g(int){}
template <> void A<int>::g(int){} // error - not something that
                                  // can be specialized
```

Answer: Members of explicitly specialized classes are unrelated to the members of the template that has been specialized (they need to have the same names, types, etc.). Definitions of such members use the same syntax, and follow the same rules, as the definitions of members of nontemplate classes.

Version added: 15
Version updated: 15

5.8 Question: What syntax is used to declare a template entity to be a friend?

Status: Open

A friend declaration such as `friend void f(int)` does two things: it declares the function `void f(int)` if not previously declared, and it makes that function a friend.

Note: Removal of friend injection from templates is still under discussion, but this is a separate issue as it involves friends in nontemplate contexts too.

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

This declares a guiding function `f(int)` and a normal function `g(int)`.

But what if you want to declare the template instance `f(int)` to be a friend without creating a guiding declaration? My proposed means of doing this is the following:

template ¡class T¿ void f(T); struct X friend void f¡¿(int); ;

The presence of a template argument list indicates that only function templates named "f" are to be considered. The argument list could, of course, include template argument values.

Specialization declarations would not be permitted in friend declarations, but template friend declarations would still be permitted:

template ¡class T¿ void f(T); struct X template ¡¿ friend void f(int); // error template ¡class T2¿ friend void f(T2); // okay ;

Actually, `friend void f<>(int)` is already permitted as far as I can tell, so the main purpose of this proposal would be to make clear that `template <> friend void f(int)` is not permitted.

Answer: A function or function template is made a friend using the normal declaration syntax (i.e., not the explicit specialization syntax).

Note: This proposal would remain unchanged even if guiding declarations were eliminated. The declarations that are now described as guiding declarations would become declarations of unrelated functions. In order to make a template instance a friend, one would have to use the `<>` syntax when naming the function to be made a friend.

Version added: 15
Version updated: 15

5.9 Question: What are the rules for exception specifications on explicit specializations? (see also 6.44)

Status: Open

A specialization is intended to provide an alternate definition for a template but should not affect the interface. Consequently, a specialization must have the same exception specification as the generated instance would. This would indicate that one of two rules could be adopted:

1. No exception specification is permitted on a specialization declaration. The exception specifications are determined by the template.
2. The exception specification must match the one specified by the template.

My proposed resolution is that the exception specification must match the one specified by the template. One minor advantage of this is that it makes it easier to construct code that can be compiled using either the new or old specialization rules (with apologies to those who don't like macros):

```
#ifdef USE_NEW_SPECIALIZATION
#define specialize template <>
#else
#define specialize /* nothing */
#endif

template <class T> void f(T) throw(T);
specialize void f(int) throw(int){}
```

This is also the closes approximation to the current WP rule that requires that the exception specification of all declarations of a function be the same.

Answer: The exception specification in an explicit specialization must match the exception specification of the template.

Version added: 15
Version updated: 15

5.10 Question: What is the linkage (internal vs. external) of an explicit specialization?

Status: Open

The WP currently says (14.10.5p3):

> *An explicit specialization of a function template shall be inline or static only if it is explicitly declared to be, and independently of whether its function template is.*

This may make sense for the `inline` keyword, but does not seem to make sense for `static`. Perhaps this is a vestigial feature from before the existence of explicit specializations.

An instance of a template, whether generated or explicitly specialized, should have the same linkage as the template itself.

Answer: The linkage (internal vs. external) of an explicit specialization is the same as the linkage of the template with which it is associated.

Version added: 15
Version updated: 15

Other Issues

6.30 Question: When is a template argument list required in a function declaration?

Status: Approved in Santa Cruz

When the requirement that specializations be declared before use was added, a new specialization syntax was added for use in explicit specializations and explicit instantiations. The new syntax was:

```
void f<>(int);  // explicit specialization
template f<>(int);  // explicit instantiation
```

In this syntax, the `<>` was needed to distinguish a specialization from a normal function declaration. Recently, the explicit specialization syntax was changed to

```
template <> void f(int);  // explicit specialization
```

which no longer requires the `<>` in the declarator.

Answer: A template argument list is permitted, but not required, in an explicit specialization and an explicit instantiation.

Version added: 14
Version updated: 14

6.31 Question: Is a template argument list permitted in a function template declaration?

Status: Approved in Santa Cruz

```
template <class T> void f(T);  // normal declaration
template <class T> void f<T>(T);  // is this permitted?
```

Answer: No.

Version added: 14
Version updated: 14

6.32 Question: Can compiler-generated functions be explicitly specialized or instantiated?

Status: Approved in Santa Cruz

Answer: No. Only user-declared functions can be explicitly specialized or instantiated.

Version added: 14
Version updated: 14

6.33 Question: When is a nested-name-specifier allowed in the declarator in an explicit instantiation.

Status: Approved in Santa Cruz

```
namespace N {
  template <class T> class A {
    void f();
  };
  template <class T> void f(T){}
  template A<int>::f();    // okay
  template N::A<int>::f(); // not allowed
  template N::f(int);      // not allowed
}
template N::A<int>::f(); // okay
template N::f(int);      // okay
```

Answer: A nested-name-specifier is allowed in the declarator in a explicit instantiation directive for a class member or a namespace member outside of its namespace. These are the same rules as when a nested-name-specifier is allowed in a normal function definition.

Version added: 14
Version updated: 14

6.34 Question: Can an explicit instantiation that refers to a class be used to instantiate all the members of a nested class?

Status: Approved in Santa Cruz

In the following example, it it possible to use an explicit instantiation directive to instantiate all the members of `A<int>::B`, or must the class referred to in an explicit instantiation refer to a "top level" template entity like `A<int>`?

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

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

template class A<int>::B;
```

Answer: Yes, an explicit instantiation directive may name a nested class within a template class.

Version added: 14
Version updated: 14

6.35 `typename` syntax problems.

Status: Option 2 approved in Santa Cruz.

There are a few problems with the current `typename` syntax.

First, there is no way to use `typename` in a using-declaration.

```
template <class T> struct A : public T {
        typename T::X x;  // Declares a member "x" of type T::X
        using T::X;  // Introduces X as a nontype
        using typename T::X;  // Not permitted by the syntax
};
```

Second, because `typename` can also be used as an alternative to `class` in a template parameter list, we have a new ambiguity between a template type parameter declaration and a template nontype parameter declaration:

```
template <class T, typename T::X x> struct B {};
```

Note that the presence of the parameter name following `T::X` cannot be used to disambiguate, because unnamed parameters are permitted.

Option 1: Both of these problems can be solved, using a suggestion made by Sean Corfield, that `typename` be changed to work the way that `template` does when used for disambiguation. The first example above would then be rewritten as:

```
template <class T> struct A : public T {
        using T::typename X;
};
```

The second example would no longer be ambiguous.

Option 2: If option 1 is too extreme at this point in the process, an alternate solution would be:

- Modify the syntax to allow `typename` in using-declarations.
- Distinguish the two uses of `typename` in a template parameter list by seeing whether the name that follows `typename` is qualified or not. When `typename` is used to specify that a name is a type, it must be followed by a qualified name. A type parameter declaration cannot use a qualified name.

Option 3: A third alternative, which eliminates the need to disambiguate template parameter declarations would be:

- Modify the syntax to allow `typename` in using-declarations (same as option 2).
- Disallow `typename` as a synonym for `class` in a template parameter declaration.

Version added: 14
Version updated: 14

6.36 Question: Where is `typename` permitted?

Status: Approved in Santa Cruz

The WP places constraints on where the typename specifier can be used, as shown in the following text from the WP:

14.2 Name Resolution

...

2 In a template, any use of a qualified-name where the qualifier depends on a template-parameter can be prefixed by the keyword typename to indicate that the qualified-name denotes a type.

3 ... The qualified-name shall include a qualifier containing a template parameter or a template class name.

The difference in wording between the two paragraph leads to questions such as whether the qualifier must truly depend on a template parameter or whether any template class name (including ones that refer to user specializations) is permitted.

I think the wording should be relaxed to allow typename to be used before any qualified name. To illustrate why, consider the following example:

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

struct AA {
        struct B {};
};

template <class T> struct C {
        typedef A<T> my_a;
        typename my_a::B b;
};
```

This is already questionable, because it is not clear whether `my_a` meets the requirement of paragraph #2 that the qualifier depend on the template parameter. Likewise, paragraph #3 requires that the qualified-name contain a template parameter or template class name. At the very least, these paragraphs would need to be changed to refer to the type specified by the qualifier and not the qualifier itself.

But what if class `C` is changed to the following?

```
template <class T> struct C {
        typedef AA my_a;
        typename my_a::B b;
};
```

It should be possible to write code using the typedef `my_a` without knowing whether or not it refers to a template parameter dependent class. You would, of course, need to use typename if `my_a` *might* refer to a template dependent class. But requiring it *only* when `my_a` refers to a template dependent class seems unnecessary.

I'm assuming that typename would still only be permitted in template contexts. This could be relaxed further by permitting typename to be used anywhere (i.e, even in nontemplate classes and functions).

Answer: `typename` may be used before any qualified name within the scope of a template declaration.

Version added: 14
Version updated: 14

6.37 Question: Does `typename` affect name lookup?

Status: Approved in Santa Cruz

I ran into some code that used `typename` that expected it to restrict the lookup to only include types. That is, in the following example, they expect the lookup of `T::X` to find the struct and not the int.

```
struct A {
        struct X {};
        int X;
};

template <class T> class B {
        typename T::X ta1; // allowed?
};

B<A> b;
```

Answer: No. `typename` is used to permit syntax analysis of template definitions, and acts as an assertion that during an actual instantiation the named entity must be a type. It does not affect the way that names are looked up, however.

Version added: 14
Version updated: 14

6.38 Question: Clarification of interaction of namespaces and specialization

Status: Approved in Santa Cruz

If a template is declared in a namespace, but its specializations also be declared in the namespace before being defined outside of the namespace? What about guiding declarations?

```
namespace N {
        template <class T> void f(T);
        template <> void f(char);
}

template <> void N::f(char);  // error - redeclaration outside
                              //         of namespace
template <> void N::f(char){} // okay
template <> void N::f(int);  // error - must be declared in namespace
void N::f(char);             // error - must be declared in namespace
```

Answer: A specialization must be declared in the namespace of which it is a member. Once so declared, it may be defined either in the namespace in which the template is declared, or in an enclosing namespace (i.e., wherever a definition of a template declared in a namespace is allowed).

A guiding declaration may only appear within the namespace in which the template is declared, because it actually adds a declaration to the namespace.

Version added: 14
Version updated: 15

6.39 Correction of default template argument description.

Status: Approved in Santa Cruz

The WP currently says (14.7 [temp.param]): The set of default template-arguments shall be provided by the first declaration of the template in that unit.

This is incorrect. It appears that a previous issue from this list was incorporated into the WP incorrectly.

The correct rules are (from issues 1.1 and 1.2 of this paper):

1. Default template arguments are permitted on class template declarations and definition.
2. The defaults need not be specified on the initial declaration.
3. After merging the default arguments from multiple declarations, the last parameter with a default argument may not be followed by a parameter without a default.
4. Default template arguments are not allowed on function template declarations, or declarations of members of class templates.

The rule about providing defaults on the initial declaration of a template actually applies to function parameter default arguments not template parameter default arguments. The rule, from issue 3.13 is: Default function arguments may only be specified in the initial declaration of a template function. This means that default arguments for member functions of class templates must be specified in the class definition and not on definition of members that appear outside of the class definition.

Answer: Core-3 decided to adopt the simpler rule that was inadvertently incorporated into the working paper. Consequently, the set of default template-arguments shall be provided by the first declaration of the template in that unit and default template arguments are not allowed on function template declarations or declarations of members of class templates.

Version added: 14
Version updated: 15

6.40 Clarification of access checking in explicit instantiation directives.

This issue and its resolution are from Bill Gibbons' reflector posting `c++std-ext-3258`.

Status: Approved in Santa Cruz

Bill Gibbons raised the issue that it is not possible to explicitly instantiate templates where the template arguments or other components of the explicit instantiation directive reference types that are not accessible.

```
namespace N {
        template <class T> void f(T);
}

namespace M {
        class A {
                class B {};
                void f() {
                        B b;
                        N::f(b);
                }
        };
}

template void N::f(M::A::B);  // should be allowed
```

Answer: The following is the wording suggested by Bill Gibbons to correct this problem, to be added at the end of 14.4 [temp.explicit]. I have modified Bill's suggested wording somewhat. My additions are shown in italics.

> The usual access checking rules do not apply to explicit instantiations. In particular, the *template* arguments, *and names used in the function declarator (e.g., including parameter types, return types, and exception specifications)* may be private types or objects which would normally not be accessible and the template may be a member template or member function which would not normally be accessible.

Version added: 14
Version updated: 14

6.41 Linkage consistency rules for specialization and guiding declarations.

Status: Approved in Santa Cruz

Answer: Core-3 decided that a specialization or guiding declaration may have different linkage than the template with which it is associated, but cannot have C linkage. As with all other cases, linkage other than C and C++ is implementation defined.

```
template <class T> void f(T);
extern "C" {
        template <> void f(int);  // not allowed
}
extern "C" void f(double);  // not allowed
```

Version added: 14
Version updated: 14

6.42 Clarification of rules for template operator new and delete.

Status: Open

In issue 6.6 (from version 4) it was decided that only the multiple parameter version of operator new could be a template. Since then we have added placement operator delete and member templates.

```
        example
```

Answer: Only the multiple parameter operator new and operator delete routines may be declared as templates. In operator new templates the first parameter must be of type `size_t`, and the return type must be `void*`. In operator delete templates the first parameter must be `void*` and the return type must be `void`.

Version added: 15
Version updated: 15

6.43 Clarification of rules for the number of things declared in a template declaration.

Status: Open

Issue 6.10 (from version 6) specified that a template declaration could not declare more than one thing. The purpose of this issue is to indicate that this also applies to explicit specializations and instantiations.

```
template <class T> void f(T), g(T);  // Error (from 6.10)

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

template <> class A<char>;  // okay

template <> void f(int), f(char);  // Error
template void f(int*), f(char*);  // Error

template <>
class A<int> f(class A<int>);  // okay -- unneeded use of
                               // elaborated type specifier

template <> class A<int> *p; // error -- "p" is not something that can
                             // be specialized
```

Answer: In a template declaration, explicit specialization, or explicit instantiation, at most one declarator may be present. When such declarations are used to declare a class, no declarator is permitted. In other words, the only forms permitted to declare a class are: "`class` *class-name*`;`" and "`class` *class-name* `{};`".

Version added: 15
Version updated: 15

6.44 Question: What are the rules for exception specifications on explicit instantiations? (see also 5.9)

Status: Open

An explicit instantiation is more like a reference than a declaration, so it seems undesirable to permit or require exception specifications to be provided.

Furthermore, it is desirable to ensure that explicit instantiation directives can be easily created by tools from information such as a list of undefined symbols produced by the linker.

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

template <> void f(int);  // okay
template <> void f(int) throw();  // error
```

Answer: Exception specifications are not permitted on explicit instantiation directives.

Version added: 15
Version updated: 15

6.45 A proposal to eliminate guiding declarations.

Status: Open

There is an existing problem with guiding declarations. The problem has been discussed frequently on the reflector.

The problem is that any "normal" function that has a type that happens to map onto an instance of a function template is considered a guiding declaration. The rules for guiding declarations state that a definition cannot be provided for such a function. So, if you accidentally happen to declare such a function, you are out of luck.

```
template <class T> T max(T,T);
int max(int,int);  // guiding declaration
```

Some historical information about how this situation came about might be helpful. Under the ARM description of templates, the following program is well formed:

file1.c:

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

file2.c:

```
void f(int){}
```

The call of `f(1)` in file1.c would call `f(int)` defined in file2.c.

This made it difficult to support certain kinds of instantiation mechanisms, so the committee added the requirement that a specialization be declared before it is used in a given translation unit. The committee also added a new syntax for declaring and defining specializations.

With this change, he call of `f(1)` in file1.c would call `f(int)` generated from the template. The definition of `f(int)` in file2.c is now ill-formed. It is only ill-formed if `f(int)` is instantiated or explicitly specialized somewhere in the program.

If we modify this example to add a third file which calls `f(int)` in file2.c, we end up with the following:

file1.c:

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

void f(int);

int main()
{
        f('c');
}
```

file2.c:

```
void f(iint){}
```

file3.c:

```
void f(int);

void g()
{
        f(1);
}
```

This program is well-formed, but

- there is no way to call f(int) in file2.c from file1.c. Any attempt to do so would result in a call of f(int) generated from the template and would render the program ill-formed.
- Any call of a the template-based f(int) in file1.c or anywhere else in the program would render the program ill-formed.

As bad as this is for normal functions, it is much worse for operator functions. Consider the impact of adding a declaration such as

```
template <class T> T operator+(T,T);
```

This changes any existing declarations of operator+ functions into guiding declarations, so a class that says

```
friend X operator+(X,X);
```

has now declared a guiding function. The function must have been defined somewhere else, so that program is now ill-formed.

Now, with member templates, we have to decide whether to

1. extend this feature to include member templates
2. have member templates and nonmembers work differently
3. change how guiding functions work for nonmembers

Issues 8.1, 8.2, and 8.3 provide some reasons why we should *not* have guiding declarations for member templates.

So, I would like to propose we do #3. Specifically, I would like to propose that we get rid of guiding functions altogether, and simply state that it is permitted to have normal functions with the same type as a potential (or actual) template instance. If you want to have a guiding function, you can simply write a normal function that calls the template. For example:

```
inline int max(int i1, int i2) { return max<>(i1,i2); }
```

This has the additional benefit that it lets you write a general template, but still lets you use specific functions written in C or assembler for specific versions. For example, the following would be permitted:

```
template <class T> T sqrt(T){ /* ... */ }
extern "C" double sqrt(double);
```

Answer: A normal function can have the same type as a potential or actual template instance. Such a function has no relationship to the template.

Version added: 15
Version updated: 15

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, except for the template parameters of the template declaration, must refer to the same entities in each translation unit. If two templates are intended to declare the same entity, but violate this rule, the results are undefined.

Version added: 15
Version updated: 15

6.47 Question: When are friend functions defined in class templates evaluated?

Status: Open

Member functions of class templates are only instantiated if they are used. This permits the user to supply a template body that would be ill-formed if instantiated for a particular template argument, but well formed for other template arguments.

Friend functions defined within a class template do not receive this special treatment. For example, `A<void*>::f` would be invalid if instantiated. But because it is not used, it is not instantiated. `f(A<void*>)` causes the program to be ill-formed even though it is not used.

Of the compilers I tried this on (EDG, Sun, cfront, g++, Borland, Watcom, Microsoft), Sun did not evaluate f(A¡void*), all the others did evaluate it and issue an error (except that one of the compilers gave an internal error).

```
template <class T> struct A {
        T t;
        int f() { return t * 2; }
        friend int g(A<T> at) { return at.t * 2; }
};

int main()
{
        A<int> ai;
        A<void*> av;

        ai.f();
        g(ai);
}
```

Answer:

Options:

1. Status quo: friend functions are always evaluated during the instantiation of a class template.
2. Friend functions defined within a class template are only evaluated if the function is used.

Version added: 15
Version updated: 15

6.48 Question: Are template friend declarations permitted in local classes?

Status: Open

Answer: No. They are pointless, so there is no reason to permit them. No other template declarations are permitted in local classes, it would be a simplification to ban all template declarations (i.e., including friend templates) from local classes.

Version added: 15
Version updated: 15

Erwin Unruh's Issues

Many thanks to Erwin Unruh who provided the following issues in finished Latex form! These issues were added to this document in version 10.

7.4 What is the point of instantiation really? (ext-2547, Erwin Unruh)

Status: Approved in Tokyo

Answer: The point of instantiation is the point of use, except that local scopes are not considered for name lookup and name injection.

Discussion: The present rules for the template name binding have a uncomfortable bit. Consider the following example:

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

void h()
{
    extern void g(char);
    f('a');          // error
}

// \#1

void g(int i)
{
    f(i);            // error ??
}
```

With the present rules both instantiations fail. The first `f<char>` should fail, since no `g` is in (global) scope at the point of instantiation and the local one is ignored (with very good reason).

The second however is not so clear cut. The WP says the point of instantiation is #1 and there is no `g` in scope. On the other hand one could argue that the function `g` is known at the call as it is not local.

This topic is currently (Nov. 1994) still under discussion and should be reviewed in a later version. It also interacts with the problem of name injection.

Version added: 10
Version updated: 10

Member Template Issues

8.1 Question: Can normal members coexist with member function templates that could generate the same signature?

Status: Open

Issues 8.2 and 8.3 illustrate cases in which it is desirable (or possibly necessary, depending on the resolution of those issues) to be able to have such coexistence. If a potential instance is allowed to coexist with a normal member function, can an actual instance do so?

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

template <> void A<int>::f(int) {}  // nontemplate member
template <> void A<int>::f<>(int) {}  // template member
```

Answer: Yes, a normal member can coexist with a member function template that could generate the same signature. It may also coexist with an actual instance of that template. The template can still be used by supplying the `<>` used to provide an explicit template argument list to the template. If the `<>` is not provided, the nontemplate member is used.

This implies that there is no such thing as a member guiding declaration.

Version added: 15
Version updated: 15

8.2 Clarification of rules for member templates and virtual functions.

Status: Open

The WP says that a member function template cannot be virtual. But can it override a virtual function from a base class?

Answer: A member template does not override a virtual function from a base class. The template is can be used to generate a function whose type matches the virtual function from the base class, but it is not virtual, and does not override the virtual function from the base class. An overriding function with a type that matches an instance of the template can be written in the derived class (and that function can explicitly call the template, if that is what is desired).

```
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
};
```

Version added: 15
Version updated: 15

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

Status: Open

```
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

8.4 Question: Can two member templates coexist whose only difference is that one is static and the other is not?

Status: Open

For nontemplate members, a static function with a given signature is not permitted to coexist with a nonstatic function with the same signature (ignoring the qualifiers on the implicit `this` parameter).

On the other hand, ambiguities between templates are, in general, deferred until use instead of being diagnosed on the declaration. Furthermore, if we wanted to apply the "existing rule" to templates, it would have to be modified to permit the declaration of `g` while disallowing the declaration of `f`.

```
struct A {
        template <class T2> void f(T2);
        template <class T2> static void f(T2);
        template <class T2> void g(T2);
        template <class T2, class T3> static void g(T2);
};

int main()
{
        void (*fp)(int) = &A::f;      // static function
        void (A::* mfp)(int) = &A::f; // nonstatic function
}

template <> void A::f(int);  // Ambiguous
```

Answer: Memebr template functions that only differ because one is static and the other is nonstatic may coexist, even though many uses of such functions will result in an ambiguity.

Version added: 15
Version updated: 15

8.5 Question: How are template conversion functions explicitly called, explicitly specialized, and explicitly instantiated.

Status: Open

```
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;
        char*  cp;
        void*  vp;

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

Answer: Template conversions functions are explicitly called, explicitly specialized, and explicitly instantiated by supplying the actual destination type as is done with nontemplate conversions.

Version added: 15
Version updated: 15

8.6 Question: Can an explicit template argument list be supplied to an constructor template or conversion template?

Status: Open

Constructors and conversion operators do not have names. Given that they have no names, it follows that an explicit template argument list cannot be supplied because there is no name to which to attach the argument list.

It might be possible to come up with some set of rules that would permit such usage, but unless there is a compelling need for this functionality, I would recommend that we simply say that template argument lists cannot be supplied for unnamed entities.

```
template <class T> struct X {};
```

```
template <class T> struct A {
        template <class U, class T> operator T*();
        template <class U, class T> A(T*);
};

int main()
{
        A<int>  ai;
        int*    ip;
        X*      xp;

        ip = ai.operator int<char>();
        ip = ai.operator X<char>();  // ??

        A<int> a2(1);  // can a template arg list go somewhere?
}
```

Answer: Template argument lists cannot be supplied for unnamed entities. This means that explicit template argument lists cannot be supplied for constructor templates and conversion templates.

Version added: 15
Version updated: 15

8.7 Question: How is a conversion function chosen when the set of conversions includes conversion templates functions?

Status: Open

The current overload resolution rules seem to adequately address the selection of a conversion function when conversion templates are included. The only piece missing is a description of what happens when more than one conversion template can produce the required type.

```
template <class T> struct X{};

struct A {
        template <class T> operator T*();
};

struct B : public A {
        template <class T> operator T();
        template <class T> operator X<T>();
};

int main()
{
        B        b;
        X<int>   xi = b;  // B::operator X<T>
```

```
        int*     ip = b;  // A::operator T*
}
```

Answer: If more than one conversion template can produce the required type, the partial ordering rules 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).

Version added: 15
Version updated: 15

8.8 Clarification of rules for standard conversions following template conversion functions

Status: Open

The working paper currently permits the "second standard conversion sequence" to be any of the ones of rank "exact match":

- No conversion
- Lvalue-to-rvalue conversion
- Array-to-pointer conversion
- Function-to-pointer conversion
- Qualification conversion

I believe that this should be further restricted to only permit the lvalue-to-rvalue conversion (and, of course, no conversion).

My understanding of the original restriction was to disallow any conversions after the template-based user-defined conversion. There doesn't seem to be any special reason why the qualification conversion should be permitted when others were disallowed.

It is useless to include the array-to-pointer and function-to-pointer conversions because the syntax does not permit such a conversion function to be written.

Answer: 13.3.3.1.2 [over.ics.user] paragraph 3 should be replaced with: If the user-defined conversion is specified by a template conversion function, the second standard conversion sequence must be either "No conversions required" or "Lvalue-to-rvalue conversion".

Version added: 15
Version updated: 15

8.9 Question: Can a member class template be declared and then defined later within the class?

Status: Open

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

Answer: Yes, as with normal nested class, member class templates can be declared and defined later within the class (or later, outside of the class).

Version added: 15
Version updated: 15

Editorial Issues

99.1 The beginning of clause 14 does not sufficiently describe the kind of template declarations that are permitted. For example, the term "template member" is not defined, and could be construed to include data members, typedefs, etc.

99.2 Nontype conversions (from issue 2.6) are not described in the WP.

99.3 Template syntax does not support an empty template argument list (`<>`).