X3J16/96-0023
WG21/N0841

Template Issues and Proposed Resolutions
Revision 14

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

September 26, 1995

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.

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.

2.29 Can a non-autonomous nested class be specialized?

2.30 Can nested classes and member template classes be specialized?

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.

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.

3.29 Template argument deduction revisited.

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)

Class Template Specific Declarations and Definitions

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)

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?

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

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

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

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

6.35 `typename` syntax problems.

6.36 Where is `typename` permitted?

6.37 Does `typename` affect name lookup?

6.38 Clarification of interaction of namespaces and specialization

6.39 Correction of default template argument description.

6.40 Clarification of access checkin in explicit instantiation directives.

6.41 Linkage consistency rules for specialization and guiding declarations.

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

7.4 What is the point of instantiation really?

7.5 Short addition to 3.17 (closed in version 13)

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

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: Open

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.

Version added: 14
Version updated: 14

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

Status: Open

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: It is proposed that only named nested class defined in "autonomous" declarations be permitted to be specialized outside of the class. Unnamed classes, and classes defined in "non-autonomous" declarations would be instantiated as part of the instantiation of the enclosing class and so, could not be specialized later.

```
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
};
```

Version added: 14
Version updated: 14

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

Status: Open

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: Open

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: Open

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: Open.

Version added: 14
Version updated: 14

3.29 Template argument deduction revisited.

Status: Open

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: 14

Other Issues

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

Status: Open

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: Open

```
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: Open

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: Open

```
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 declaration.

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: Open

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: Open

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: Open

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: Open

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: Open

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 N::f(int);  // okay
void N::f(char);             // error - must be declared in namespace
```

Answer: A specialization may be declared or 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: 14

6.39 Correction of default template argument description.

Status: Open

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.

Version added: 14
Version updated: 14

6.40 Clarification of access checkin in explicit instantiation directives.

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

Status: Open

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: Open

Answer: I propose that the linkage of specializations and guiding declarations must match the linkage of the associated template.

```
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

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.

7.3 How does type deduction interact with conversions (ext-2320, Erwin Unruh)

Status: Open

At the moment I see the following problems, where templates enter the discussion of conversions.

1. Conversion from pointer to derived to pointer to base:
   Both the derived and the base class could be template classes. At this point there should be no big problem. Both classes must be complete to allow such a conversion. The base class must be instantiated for the derived class to be defined. The derived class must be instantiated whenever a pointer to it is subject to a conversion. (see point 2.7)
2. Conversion of pointers to member
   When solving the conversion of template arguments we left out member pointer. So pointer to member conversions cannot interfere with template type deduction. So source and target of such a conversion are fixed and it can be checked whether the types are completely defined.
   Proposal: When a pointer to a member of a template class may be the target of a conversion, that class will be instantiated.
3. Constructor templates
   This does have a very neat solution after the proposal for the section 13 is accepted (94-0080). Here the overload resolution goes back to itself whenever a user defined conversion comes into play. So the conversions itself are described using overload resolution. In this context it is relatively easy to incorporate constructor templates into that scheme.

4. Conversion templates

   The usage of conversion templates is discussed in Point 7.1. There is however an additional problem in the declaration matching. Consider the following example:

```
class A {
    operator int();
};
class B : public A {
    template <class T> operator T ();
};
class C : public B {
    operator char ();
};
```

   The question is whether the template hides the conversion in the base class and whether a declaration in the derived class may hide (an instance of) the template. The problem arises since the return type of the conversion operator is considered its name.

   Proposal: The template conversion does hide only the conversions which have an exact match. A program is ill-formed, if a conversion template is potentially hiding (or being hidden by) a conversion for which the type deduction can not be done.

   To elaborate that rule: If there is a potential hiding between a template and a normal conversion, try type deduction. If that results in a match, fine. If the result is that there is definitely no match (`int` vs. `T*`), fine. Otherwise, there is a problem!

   Hiding between two template conversions should be discussed when the topic of partial specialization is resolved. Is it allowed to have two template conversions in the same class ?

Version added: 10
Version updated: 10

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

Status: Tentatively approved in Monterey.

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

Editorial Issues

8.1 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.

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