X3J16/95-0007
WG21/N0607

Template Issues and Proposed Resolutions
Revision 11

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

January 31, 1995

Revision History

Version 1 – March 5, 1993: Distributed in Portland and in the post-Portland mailing.
Version 2 – 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 – September 28, 1993: Distributed in pre-San Jose mailing. Reflects decisions made in Munich. No new issues were added in this revision.
Version 4 – 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 – 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 – 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 – 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 – 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 – 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 – Novermber 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 – January 31, 1995: Distributed in the pre-Austin mailing. Includes a few 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.

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)

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?

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.

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.

3.22 Proposed rules for selecting between overloaded function templates

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?

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.

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?

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?

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 probems

Erwin Unruh's Issues

7.1 Type deduction for conversion operators

7.2 How does type deduction interact with overloading

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

7.6 Type deduction with several results

Nontype Parameters for Function Templates

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

Function Templates

3.7 Question: Can template parameters be used in qualified names in function template declarations?

```
template <class T> struct A {
        struct B {
                friend B& operator +(const B&, const B&);
        };
};

template <class T> A<T>::B& operator +
     (const A<T>::B& b1, const A<T>::B& b2){}

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

int main()
{
        A<int>          a;
        A<int>::B       b1;
        A<int>::B       b2;
        A<int>::B       b3;
        f(a, b1);
        b1 = b2 + b3;
}
```

There are two issues involved here

1. how does one specify that a name like `A<T>::B` is a type?
2. can `T` in `A<T>::B` be deduced?

The first issue is discussed, and a proposal made in 94-0191/N0578 "Major Template Issues, Revision 0", and will not be discussed here.

The remainder of this discussion focuses on the type deduction issue. This deduction has deadlocked in the past, partially on the lack of substantial justification for this feature.

I have discussed this issue with Alex Stepanov who informed me that not only it it important for STL, but the current implementation of STL has had to use member functions where friend operator functions are really needed, because the compiler being used to develop STL does not support this feature.

The question really boils down to "can nested types be used in function template declarations". The arguments for supporting this kind of usage are the same as the arguments for providing nested types at all. In my opinion, it should be possible to take just about any class and convert it into a template. Banning nested types in function template declarations would make it impossible to convert many kinds of classes into template equivalents.

There are at least two compilers (IBM and EDG) that currently support this feature.

Note that now that nontype template parameters may be used in function templates, the same principle applies to nontype parameters. For example,

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

template <int I> void f(A<I>, A<I>::B){}
```

One concern that has been expressed regarding this feature is that in a construct such as `T::A`, `T` is the class in which `A` is declared and not strictly a type attribute of `A`. While this is true, it does not change the fact that what is being deduced is in fact a type (or nontype in the case of nontype parameters). The question is whether the class of which a type is a member can be used as information from which type (or nontype) information is deduced. In other words, we are not adding a new kind of deduction, we are simply expanding the kind of information that can be used by the deduction process.

Answer: Yes, this kind of deduction is allowed.

Note that the type of the actual argument must be a nested type (class/struct, union, or enum). A typedef is simply a synonym for another type and cannot be used.

This proposed resolution suggests that a compiler should be able to determine that names used in this context are types. An alternative would be to require explicit designation as a type. The current facility for such designation (using `typedef`) is not well suited for this kind of construct, so some change to the current facility would probably be required.

```
template <class T> struct A {
        typedef T::X;
        T::X x;
        T::X f();
        friend void g(T::X);
        friend void g(T::X2);  // Error
        template <class U> void h(U::Y);  // OK
};

template <class T> T::X A<T>::f(){}  // OK
template <class T> void g(T::X);  // OK
```

Status: Open

Version added: 1
Version updated: 7

3.13 Revisiting default arguments.

I would like to recommend that we revisit the proposed rules for default arguments to restrict the ways in which default arguments for function templates may be modified.

This is motivated by examples such as the following. If it is possible to add default arguments to a function template with template parameters that depend on other template parameters, then the new default argument would need to be type-checked for each of the instantiations that have already been generated – a process which has the potential of yielding new errors for the already generated instantiations.

While this is possible to do, I think it would be more confusing to users than a simpler restriction on how default arguments may be modified once a tempalte is declared. I would recommend the same for member functions.

```
template <class T>
void f(T, T, T*);  // Default arg information locked here

void g1()
{
        int i;
        f(i,i,&i);
}

template <class T>
void f(T, T, T* = new T);  // Error - default arguments
                           // modified after the first use

void g2()
{
        int i;
        f(i,i);  // Without this rule, is this legal?
        char c;
        f(c,c);  // How about this?
}
```

In the following example, a default argument is provided that is only valid for certain instantiations. How would the behavior of this program change if the default argument declaration (currently declared at point #2) were moved to either #1 or #3?

If the declaration were at point #1, an error would be issued at the call labeled #4 because the default argument is incompatible with the parameter type.

If the declaration were at point #2, should an error be issued at point #2 because the default argument is invalid for an existing instantiation? Or, should the error only be issued if the default argument value is actually used in an invalid call?

Unless we adopt a rule that prohibits changing the default arguments once name binding has occurred (at the latest), we introduce a situation in which the legality of one call depends on whether or not a previous call of the same function has been seen. I think this is undesirable.

```
template <class T> void f(T, T);
struct A {};
// template <class T> void f(T, T = 1);  // #1
```

```
void g1()
{
        int i;
        A a;
        f(i,i);
        f(a,a);  // #4
}

template <class T> void f(T, T = 1);  // #2

void g2()
{
        int i;
        A a;
        f(i);
        f(a,a);  // Is this an error?
        f(a);  // Error: default argument has wrong type
}
// template <class T> void f(T, T = 1);  // #3
```

Answer: Default 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.

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

template <class T>
void f(T, T = 1){}  // Error - default not on initial declaration

template <class T> struct A {
        void f(int);
        void g(int = 0);
};
template <class T> void A<T>::f(int = 0){}  // Error
template <class T> void A<T>::g(int){}  // OK
```

Status: Tentatively approved by the Extensions working group in Valley Forge.

Version added: 5
Version updated: 10

3.21 Question: 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.

```
file1.c:
-------
template <class T> void f(T);  // #1
void f1()
```

```
{
        int *ip;
        f(ip);  // attempted call of f(int*) generated from #1
}

file2.c:
-------
template <class T> void f(T*);  // #2
void f2()
{
        int *ip;
        f(ip);  // attempted call of f(int*) generated from #2
}

file3.c:
-------
void f(int*);  // #3
void f3()
{
        int *ip;
        f(ip);  // attempted call of nontemplate f(int*)
}
```

Another way to put this is "What is the signature of a template function?" The two most obvious alternatives are

1. A template function signature includes the function name and function parameter information, but only includes information about template parameters that cannot be deduced from the function parameter information. This model would not allow templates #1 and #2 to coexist. Depending on how it is defined, it may or may not allow template functions and nontemplate functions (#3) to coexist.

   Although it is not really clear in the WP, this is probably what most people would consider the status quo.

2. A template function signature includes the function name, function parameter information, and information about all of the the template parameters. This model would allow all of the functions in the example above to coexist.

One of the characteristics of the newly adopted compilation model (and on of its drawbacks, in my opinion) is that a processor must be able link a reference to a template function with the body of the associated function template definition at the point at which all of the translation units are brought together to form a complete program. This requires that we either select a model in which complete information about both the template and function parameters is part of the signature, or we disallow the coexistence of function template definitions in cases in which there is any overlap between the sets of functions that could be generated by the two functions. The latter solution is most certainly two extreme as it would outlaw examples such as the following and would, by definition, make partial specialization of function templates impossible.

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

A third alternative would be to require that all template definitions be supplied at compile time. This would allow the first model to be used since it would no longer be necessary to link the reference and definition at "link time".

Revisiting the two models described at the beginning of this issue, we can select model #1 and change the compilation model or select model #2.

If model #2 is used, it must be possible to link specialization declarations with the templates that they specialize. In the following example, the first specialization declaration is ambiguous because it could be linked to either of the templates. The second declartion is legal because by specifying both the template and function parameters only one of the templates matches the declaration. Likewise, in the third declaration only one of the templates matches the function parameter list, so the call is legal.

```
template <class T> void f(T);
template <class T> void f(T*);
void f<>(int*);        // #1 - Ambiguous
void f<int>(int*);  // #2 - OK
void f<>(int);  // #3 - OK
```

Answer: TBD

Status: Open

Version added: 11
Version updated: 11

3.22 Proposed rules for selecting between overloaded function templates

This is a proposal to permit selection between a set of overloaded function templates in cases in which the calls are currently considered ambiguous because the functions are identical as far as overload resolution is concerned.

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

template <class T> void f(T, T){}          // #1
template <class T> void f(A<T>, A<T>){}   // #2

int main()
{
        int         i1, i2;
        A<int>         a1, a2;
        f(i1, i2);  // calls f(T,T)
        f(a1, a2);  // currently ambiguous --
                    // should call f(A<T>, A<T>)
}
```

In the above example it is desirable for the call `f(a1, a2)` to call the function template that accepts an `A<T>` parameter instead of the one that simply uses a T parameter. The call is currently ambiguous, because the two potentially generated functions have the same signature. What is needed is some additional information in the overload resolution process that can serve as a tie breaker.

In cases where the tie breaker is needed, the overload resolution process will result in two or more function templates which, if instantiated, would produce identical function signatures, but would do so through the use of different templates.

To decide between the members of this set, the function parameter lists are compared. If one of the parameter lists could be transformed into the other parameter list by providing the suppling the appropriate values to be substituted for its template parameters, then the more specific of the two functions should be used. If the two function parameter lists cannot be made equivalent by this kind of substitution, then the functions are unordered relative to one another and the function call is ambiguous.

Put another way, you do template argument deduction using the more specific function as the "actual argument" types. For example, you try use f(T,T) as the function template, and f(A<T>, A<T>) as the actual argument list from which the type A<T> is deduced as the type of T. If such deduction is possible, then one call is a special case of the other.

As with overload resolution in general, top level references are removed before this comparison is done, as are any type qualifiers immediately under top level references. In other words, T& and const T& are both transformed to T before this comparison is done.

Examples:

1. When called with f(const T), T can be replaced with const T — use #2

```
f(T)          // 1
f(const T)    // 2
```

2. When called with f(const T*), T can be replaced with const T — use #2

```
f(T*)         // 1
f(const T*)   // 2
```

3. When called with f(T), ambiguous (top level references are ignored)

```
f(T)         // 1
f(T&)        // 2
```

4. When called with f(A<int>), T can be replaced with A<T> — use #2 (top level references are ignored)

```
f(T&)       // 1
f(A<T>)     // 2
```

5. When called with f(const A<int>), T can be replaced with A<T> — use #2 (top level references and associated qualifiers are ignored)

```
f(const T&)       // 1
f(A<T>)           // 2
```

6. When called with f(const A<int>), pick #1, #2 is not callable

```
f(const T&)       // 1
f(A<T>&)          // 2
```

7. When called with f(A***), T can be replaced with T*, use #2.

```
f(T*)      // 1
f(T**)     // 2
```

8. When called with `f(A<B<int>>)`, use #3 because `T` in #1 can be replaced with `A<B<T>>` and `T` in #2 can be replaced with `B<T>`.

   ```
   f(T)          // 1
   f(A<T>)       // 2
   f(A< B<T> >) // 3
   ```

9. When called with `void(*)(char **, int*)`, ambiguous

   ```
   f(void (*)(T, int))   // 1
   f(void (*)(int, T))   // 2
   ```

Status: Open

Version added: 11
Version updated: 11

Member Function Templates

4.10 Question: How are members of class templates declared and defined?

Answer:

1. A specialization of a member function of a template class (but not a member template) is declared or defined using the normal function member function syntax (i.e., without use of the `<>` used in template function specializations). Note that some additional specialization syntax such as `template <>` or `specialise` would be useful in contexts like this to make specializations easier to spot. Default arguments may be added in these declarations, but previously declared default arguments may not be modified or redeclared.
2. A member template is defined outside of the class definition in much the same way as a nontemplate member, but the template parameter list of the class is followed by a second template declaration instead of being followed by a normal function declarator. Default arguments may not be supplied in these declarations.
3. A member template of a class template may be specialized for a given instance of the class as indicated in the example below. Default arguments may not be supplied in these declarations.
4. A specific instance of a member template may be specified using the normal template function specialization syntax. The `<>` is required. Default arguments may not be supplied in these declarations.

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

// Specialization - #1
void A<int>::f(int);

// Out of class definition - #2
```

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

// Specialization of member template - #3
template <class X> void A<int>::g(int,X);

// Specialization of an instance of a member template
void A<int>::g<>(int, char);
```

Default argument rationale: It has been tentatively decided that default arguments may only appear on the initial declaration of a template. This rules out the use of default arguments on the definition or specialization of a member template (#2 and #3 above). It has further been decided that default arguments are not permitted on specialization of function templates. This rules out the use of default arguments on the declaration of a specialization of an instance of a member template (#4 above).

A specialization of a member function of a template class is different from either of these cases. It doesn't declare a template, nor is a member function of a class template really a function template (i.e., member functions of class templates do not participate in the function template matching process and in template argument deduction).

Status: Open

Version added: 11
Version updated: 11

Other Issues

6.18 Issues regarding declarations of specializations.

The language was recently revised to require that a specialization be declared before it is used. For example,

```
template <class T> void f(T){}
void f<>(int);  // Declares that a specialization of
                // f(int) will be provided
```

While this usage is clear for normal template functions, it is problematic for members of template classes. In the nonmember case shown above, the template argument list makes it clear that the function is a specialization. In the member function, only the argument list of the class is present, making the purpose of the declaration less clear. For static data members the problem is even worse because the syntax for the specialization is already used to mean a definition for which no specific value is provided.

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

void A<int>::f();  // Is this a specialization declaration?
int A<int>::i;     // This is a definition, not a declaration
```

I propose that a keyword be added to designate a declaration as a specialization and that the current syntax for specializations be eliminated. The following are some of the possible keywords:

```
template <class T> struct A {
        static int i;
};

specialize int A<int>::i;
specialise int A<int>::i;
specific int A<int>::i;
specialism int A<int>::i;  // Yes, specialism is a real word
```

Of these, I personally prefer `specialize` because it matches the wording used in the working paper. If `specialize` is not acceptable because it is spelled differently in some countries, then `specific` would probably be my second choice.

Two alternatives have been proposed that do not involve the addition of a keyword:

```
template <> int A<int>::i;
extern int A<int>::i;
```

Status: Open

Version added: 7
Version updated: 10

6.21 Question: How is a dependent name known to be a template?

This issue was raised by Erwin Unruh in `c++std-ext-2239`.

In the following example from Erwin's posting, the `f` on the indicated line refers to an integer data member in `A<B>`, and to a function template in `A<C>`.

```
template <class T> class A : public T {
        void foo(){
                T t;
                f < 1 > (t,t);        // critical line
        }
};

class B {
         int f;
};
int operator> (B, bool);
A<B> ab;

class C {};
template <int I, class T> void f(T, C);
A<C> ac;
```

In another example from Erwin's posting, a variation of the problem using member templates is illustrated.

```
struct A { int x; };
struct B { template<int> void x(int); };

template <class T> struct C : public T {
        void foo(){
                x < 1 > (2);        // critical line #1
        }
};

C<A> ca;        // #1 is double comparison
C<B> cb;        // #1 is template function
```

Answer: We currently have a means of designating that a given name is a type for use when a type will be defined in a template dependent base class. I propose a similar mechanism for templates. A name will be assumed not to be a template unless explicitly designated as one.

```
template <class T> class A : public T {
        template f;  // May be placed here
        void foo(){
                T t;
                template f;  // or may be placed here
                f < 1 > (t,t);
        }
};
```

The second example would be modified as follows:

```
struct A { int x; };
struct B { template<int> void x(int); };

template <class T> struct C : public T {
        template T::X;  // May be placed here
        void foo(){
                template T::X;  // or may be placed here
                x < 1 > (2);        // critical line #1
        }
};

C<A> ca;        // #1 is double comparison (now made invalid)
C<B> cb;        // #1 is template function
```

The identifier following the `template` keyword must either have no qualifier or have a qualifier that begins with either a template parameter or a template class name.

If this proposal is adopted, I believe we should modify one of the existing uses of the keyword `template`. It is currently used for template declarations and for explicit instantiation requests. I believe that using it for both explicit instantiation requests and for explicit template designation would be confusing. I propose that a new keyword `instantiate` be added for use in explicit instantiation requests and that the keyword `template` no longer be supported in that context.

Status: Open

Version added: 7
Version updated: 7

6.22 Interaction between templates and namespaces

The September 20, 1994 Working Paper contains a new section (with an editorial box) that specifies that an instantiation caused by a template reference within a namespace is done within the namespace that contains the reference. Prior to this change the WP specified that the instantiation was done at the global scope.

The new WP description does not address what happens if the template itself was defined in a namespace.

```
namespace N {
        void g(int);
        template <class T> void h(T);
        void i(int);
        template <class T> void f(T t)
        {
                g(t);
                h(t);
                i(t);
        }
};

namespace X {
        void g(int);
        void h(int);
// Effective instantiation point of f(int)
        void m()
        {
                f(1);  // which g(int) should be called when f(int)
                       // is instantiated?  Which version of h(int)?
        }
};
```

In the example above, the new WP wording would tend to suggest that the instantiation is done at the point indicated, suggesting that `X::g(int)` and `X::h(int)` should be called. Selecting that instantiation point would also suggest that the call of `i(int)` would be invalid because there is no function `i` in scope at the specified point.

If the indicated point is truly the effective instantiation point, we need additional name lookup rules so that names from the namespace in which the template was defined can be available when the instantiation is done.

There seem to be four possible alternatives:

1. Look first in the template definition namespace, then in the referencing namespace. In the example above, look first in namespace `N`, then in namespace `X`.

2. Look first in the referncing namespace, then in the template definition namespace. In the example above, look first in namespace `X`, then in namespace `N`.
3. Look in both namespaces and allow overloading of names between the two namespaces.
4. Look in both namespaces, if the name is found in both namespaces the program is ill formed.

Answer: Alternative number 3 approved in Valley Forge.

Status: Open

Version added: 9
Version updated: 9

6.24 Question: May function types be used as template parameters?

The use of function types as template parameters is problematic. It can result in something that looks like an object changing into a function declaration. Note that it is not possible to define a member such that it could be instantiated as either an object or a function.

```
template <class T> struct A {
        static T t;
};

A<int()> a1;  // Oops!  A<int()>::t is now a function!
```

Likewise, a functiontype inherited from a template dependent base class can create the same problem.

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

struct B {
        typedef int(T)();
};

A<B> a1;  // Oops!  A<B>::t is now a function!
```

Answer: Declaring a function or member function with a type that depends on a tempalte parameter may only be done using the syntactic form of a function declarator.

Status: Tentatively approvded by the Extensions working group in Valley Forge.

Version added: 9
Version updated: 10

6.27 Name binding problems.

As the name binding rules are currently written, object and type names are always bound at template definition time. However, this assumes that function names are not visible as the names of other entities at the time the initial name binding is done.

Example 1 illustrates how this problem can occur if a dependent function name happens to have the same name as an already declared type. Example 2 illustrates a more serious problem where a name from the defining namespace is used when a dependent name from the reference namespace is intended to be used.

Example 1:

```
struct X {};

template <class T> void f(T t)
{
        // X(t) is intended to be a dependent function
        // call but instead
        // it is interpreted as an object declaration.
        X(t);
}

void X(int){}

int main()
{
        f(1);
}
```

Example 2:

```
int X;

template <class T> void f(T t)
{
        // X(t) is intended to be a dependent function
        // call but instead
        // it is interpreted as a syntax error.
        X(t);
}

namespace Y {
        void X(int){}

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

Answer: Unknown

Status: Open

Version added: 11
Version updated: 11

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.

7.1 Type deduction for conversion operators (ext-2434, Erwin Unruh)

Since we added member templates, implicit type conversions can involve template functions. They can be constructors or type conversions.

Constructors are involved if the target type is a class. Then the source of the conversion is the argument of the constructor call and the normal rules of type deduction can be used (including some implicit conversions).

For conversion operators the case is different: Here the source is a class and the conversion operators of the class (and its bases) are considerd. Non-template conversions build a finite list and we can check them all. Template-conversions however must be sorted out before we can check them.

How do we determine the template arguments for the conversion operator? All we have as information is the target type of the conversion.

If there is an exact (really exact) match that set of arguments should build one viable conversion. How about any other (standard) conversions which could be done after the conversion operator? I consider the conversion allowed for normal type deduction:

- function ... to pointer to function ... : This should not happen, since a return type cannot be a function.
- array of T to pointer of T : This should not happen, since a return type cannot be an array.
- pointer to more qualified pointer : This may follow the same rules as for normal functions. Unless the added (or removed) qualifications do not involve the deduced type, those conversion may be done.
- pointer to derived to pointer to base, restricted form: For normal functions this conversion is allowed if it leads to an `B<something>`. This is safe because there is a finite set of bases for each class and we start at the derived end. A similar conversion in this place would be a conversion from a `D<something>` to a `B`. This is more difficult, since we start at the `B` and are trying to find a derived class. The derived class may even be forced to be generated by that conversion. Because of this differences I would disallow this sort of conversion.
- pointer to member: As this is inverted from the normal deduction, it would work. However, it would be easier to teach and understand to stay with the simple rule stated below.

See the following example:

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

template<class T> struct D : B<T> {};
```

```
template<class T> int foo( int B<T> );

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

A a;
int i = foo(a);
```

At this call, no instance of `D` exists. To check whether we can call function foo we must instantiate some classes to check, which of them will have base `B<int>`. The obvious shortcut of chosing the same template arguments won't work.

It is always a bad idea to search for derived classes and this would be such a case. We should not allow this!

A first try of a formulation would be:

In a conversion sequence containing a conversion template, for which the template parameters are deduced, the second standard conversion should only contain lvalue conversions, rvalue conversions or qualification conversions.

This excludes promotions and conversions of the category 'standard'. (the categories are from the new chapter 13, 94-0080)

Status: Open

Version added: 10
Version updated: 10

7.2 How does type deduction interact with overloading (ext-2320, Erwin Unruh)

After determining the candidate functions, for each template function type deduction takes place. That type deduction can have different results:

1. A single function is chosen.

   In this case that function is the candidate replacing the template (It will also be a viable function).

2. No match is found.

   In this case there is no viable function instance.

3. Several functions (finite number) may be chosen.

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

   template<class T> int f( A<T> );

   class B : public A<char>, public A<int> {};

   B b;

   int i = f(b);
   ```

In this example, both bases produce each a viable function from the template. The result is that this call is ambigous.

There may be situations where two viable functions may be generated from a template, one of them better than the other. (One possibility may be a specialization which is derived from another specialisation of the same template). Another sensible case would be where both functions are worse than a non-template function (add `f(B)` to the example).

Resolution: All viable functions are considered for overloading.

Alternative 1: No function produced from this template is considered for overloading.

Alternative 2: The call is ill-formed.

4. The type deduction fails.

   This can be the case when an infinite number of viable functions is generated, a template argument cannot be deduced or a conflict between deductions arises (3.16).

   In theses cases no reasonable function can be chosen from that template. (See also 7.6)

   Resolution: The call is ill-formed.

   Alternative 1: No function produced from this template is considered for overloading.

After this step of type deduction the normal overload resolution takes place. A template function should not be in disadvantage at normal overloading (remove box 59 together with the preference of non-template functions).

Comment from John Spicer: The original template overload resolution rules from the ARM favored a nontemplate exact match over a template exact match (the only kind of match allowed for templates in the ARM). When I proposed a revised set of overloading rules that allow conversions in template function calls I retained the bias for nontemplate functions). In other words, the bias has always been there and I don't think we should eliminate it without a good reason to do so.

Status: Open

Version added: 10
Version updated: 10

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

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 reolution 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 his 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 definitly no match (`int` visa `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 ?

Status: Open

Version added: 10
Version updated: 10

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

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.

Status: Open

Version added: 10
Version updated: 10

7.5 Short addition to 3.17 (ext-2455, Erwin Unruh)

I want to add a new point to the example: It reads now

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

void g(int, int);      // g1
void g(int, char);     // g2

template <class T> void h(int, T);

int main()
{
    f(g);    // O.k. chooses g1
    f(h);    // ??
}
```

The call with g is no problem. The function g2 can be no match, since its second parameter does not match the second parameter of the function pointer.

The call with h looks similar. Here is only one function, which can be the argument to the call. This is the case since the template parameters involve different parameters of the function pointer. But to solve this problem we have to do type deduction on both f and h in parallel. This looks very strange.

So I propose to add a new rule (or a variant of the one in spicer's list):

If a template argument is deduced from a function parameter of a pointer to function type, the function argument must be an expression of a single type, or a name of an overloaded function which does not contain template functions.

There is a short note in 3.17 which supports this view. It reads: "... (i.e., in which no type deduction is required)."

Status: Open

Version added: 10
Version updated: 10

7.6 Type deduction with several results (ext-2436, Erwin Unruh)

After the adopton of 3.9 (conversion of template arguments) and 3.16 (deducing from several arguments) we have a strange situation which is not handled by the rules.

See the following example:

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

class D : public B<int>, public B<float> {};

template <class T> int f ( B<T> );
template <class T> int g ( B<T> , B<T> );

D d;

int i = f(d);
int j = g(d,d);
```

Let's first look at f. The argument is a class which does not match. It has two base classes which can be reached and which would match. So there are two instances of the template which can be used for overload resolution. (In this case they are ambiguous, but another parameter could have solved the ambiguity).

The function g gets more interesting. For both parameters we can deduce the argument type for the template. Counting all tuples they colud be:

1. int and int
2. int and float
3. float and int
4. float and float

Only the first and last possibilities really lead to correct functions.

If we are going to allow several functions, we have to describe a complete algorithm of how to find the viable functions.

I propose to have a simple rule, which may be different from ordinary functions. The rule is: Type deduction from a single argument must lead to at most one choice of a template argument. If two different argument values can be deduced, the call is ill-formed. (See also 7.2)

I am not very firm on this conclusion and would like to hear more arguments!

Status: Open

Version added: 10
Version updated: 10