X3J16/94-0096
WG21/N0483

Template Issues and Proposed Resolutions
Revision 7

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

June 1, 1994

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.

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

1.10 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.

2.23 How does name injection interact with the semantics of friend declarations?

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?

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

3.16 Clarification of template parameter deduction rules.

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

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

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

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?

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

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

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?

6.18 Issues regarding declarations of specializations.

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

6.20 Template compilation model proposal.

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

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

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

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

Status: Open

Version added: 7
Version updated: 7

2.23 Question: How does name injection interact with the semantics of friend declarations?

The semantics of friend declarations in class templates are not clear. Prior to adoption of the "no injection" rule, most implementations seem to treat a function declared in a friend declaration in a manner similar to the equivalent declaration appearing outside of the template. The declaration, in addition to granting friendship, also affected overload resolution. In the following example `f(A<int>,int)` is injected from the instantiation of `A<int>`. As a result, the call of `f(ai, 'c')` is treated differently than the call of `f(1, 'c')` because the former allows conversions of its arguments while the latter does not[1].

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

template <class T> struct A {
        friend void f(A<T>, int);
};

int main()
{
        A<int> ai;
        f(ai, 'c'); // Calls f(A<int>,int) with conversion
        f(ai, 1);   // Calls f(A<int>,int)
        f(1, 'c');  // Error - no matching function
        f(1, 1);    // Calls f(int,int)
}
```

[1] This example is written using the "old" rules that permitted name injection and also the rules that prohibited conversion of function template arguments. Under the new conversion rules, the conversion of `char` to `int` would be allowed even for the function template call.

The motivation for the "no injection" rule was to avoid the silent introduction of declarations that affect overload resolution, so clearly the old semantics of friend declarations are made obsolete by the no injection rule. But what are the new semantics?

Proposal one: A friend declaration in a template class conveys friendship, but does nothing more. It has no effect on overload resolution or on the source of the definition. A specialization declaration may not be used in a friend declaration. A class specialization is treated as a normal class declaration for purposes of injection.

This proposal has a number of unfortunate consequences. A friend declaration in a class template is unlike any other declaration in the langauge. Furthermore, when selecting rules for class specializations one must choose between the normal class rules and the class template rules. Either choice results in some kind of inconsistency. Either it is impossible to write a class specialization whose semantics duplicate those of the class template or it is impossible to write a class specialization (or class template) whose semantics duplicate those of a normal class.

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

struct A {
        friend void f(A, int); // Friendship & overload resolution
        friend void f<>(A int); // ???
};
template <class T> struct B {
        friend void f(B<T>, int); // Friendship only
        friend void f<>(B<T>, int); // ???
};
void f(B<int>, int);  // Affects only overload resolution
void f<>(B<char>, int); // Affects only source of definition
```

Proposal two: Proposal two is a proposal to bring back name injection. John Barton and Lee Nackman of IBM have a paper in the pre-Waterloo mailing[2] arguing that injection from templates is an important facility. As a consequence of the Barton/Nackman paper the injection issue may be revisited. Should the committee's previous decision regarding injection be reversed I want to have an equivalent clarification of friend semantics available as part of such a resolution.

The first part of this proposal is a description of the proposed name injection rules (this was the original version of issue 2.10 from revision 1 through revision 4 of this paper).

```
// Name injection
template <class T> struct A {
        friend void f(A<T>){}
        friend void f2(struct X* x);
};

void main()
{
        void* fp;
```

[2] I've been told that such a paper is expected to be in the mailing

```
        X* x;   // Error - X is undefined
        fp = f; // Error - f is undefined
        f2(x); // Error - f2 is undefined
        A<int> a;
        X* x2;  // OK - X defined during instantiation of A<int>
        fp = f; // OK - only one instance of f
        A<char> ac;
        fp = f; // Error - f is now overloaded
}
```

Nothing is injected when the class template is scanned. `X`, `f(A<int>)`, and `f2` are injected into the global scope when `A<int>` is instantiated. When `A<char>` is instantiated, `f(A<char>)` is injected into the global scope. `X` already exists so nothing else is done with `X`.

The second part of the proposal clarifies the sematics of friend declaration in template classes.

A friend declaration in a template class, as in other classes, conveys friendship and injects a declaration into the enclosing scope that affects overload resolution. A specialization declaration may not appear in a friend declaration.

The advantage of this proposal is that normal classes, class temlates, and class specializations are all handled in the same way.

Status: Open

Version added: 7
Version updated: 7

Function Templates

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

```
template<class T> void f(T::X a);
template<class T> void g(T::E a);
template<class T> void h(T::I a);

struct X {
        struct Y {};
        enum E {};
        typedef int I;
};

struct Z {
        typedef X::Y my_y;
};

void g()
```

```
{
        X::Y y
        X::E e;
        X::I i;
        Z::my_y y2;
        f(y);  // f(X::Y);
        f(y2); // f(X::Y);
        g(e);  // g(X::E);
        h(i);  // Error - type of i is int not X::int
}
```

The example shown above illustrates the most general use of this construct. A more typical use would probably be something like the following:

```
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;
}
```

This illustrates that 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. A name declared this way is assumed to be a type name.

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.

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 specify that the default arguments for a given specialization be locked in at the point that name binding occurs.

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 simply saying that the default argument information is locked in when the first instance of the template is referenced. I would recommend the same for member functions.

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

void g1()
{
        int i;
        f(i,i,&i);  // Default arg information locked here
}

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, 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
```

Status: Open

Version added: 5
Version updated: 7

3.14 Question: What are the rules regarding use of the inline keyword in function template declarations?

Answer: Whether a function template is declared as being inline or static has no effect on specializations. If a specialization is to be inline it must be declared inline regarless of how the template was declared.

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

inline void f<>(int){}   // OK
void g<>(int){}          // OK (not inline)
```

Declarations of any given template or specialization must be consistent with previous declarations (using the same rules that apply to nontemplate functions).

```
template <class T> void f(T) {}       // Defaults to noninline
template <class T> inline void f(T);  // Error: conflicts with
                                      // previous declaration

template <class T> inline void g(T) {}
template <class T> void g(T);  // OK - defaults to previous
                               // declaration
```

Status: Open

Version added: 7
Version updated: 7

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

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

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

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

        f(u);   // Calls f(union T)
        f(s);   // Calls f(struct T)
        f(c);   // Calls f(struct T)
        f(e);   // Calls f(enum T)
}
```

Answer: An elaborated type specifier in a function template declaration restricts the function matching process so that only actual arguments of the appropriate kind will match the template.

Status: Open

Version added: 7
Version updated: 7

3.16 Clarification of template parameter deduction rules.

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

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

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

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

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

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

int main()
{
        A       a;
        B       b;

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

Status: Open

Version added: 7
Version updated: 7

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

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

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

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

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

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

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

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

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

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

Status: Open

Version added: 7
Version updated: 7

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

file1.c:

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

file2.c:

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

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

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

Status: Open

Version added: 7
Version updated: 7

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

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

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

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

Status: Open

Version added: 7
Version updated: 7

Member Function Templates

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

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

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

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

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

Status: Open

Version added: 7
Version updated: 7

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

Answer: Yes. (However, see also 6.18)

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

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

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

Status: Open

Version added: 7
Version updated: 7

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

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

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

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

Answer: No

Status: Open

Version added: 7
Version updated: 7

Other Issues

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

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

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

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

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

Status: Open

Version added: 7
Version updated: 7

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.

Status: Open

Version added: 7
Version updated: 7

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

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

`typedef` *qualified-name*;

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

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

Status: Open

Version added: 7
Version updated: 7

6.20 Template compilation model proposal.

One of the issues that has been around for a long time is how a user needs to organize his/her template source code so that it may be moved from one compiler to another without requiring any reorganization.

The specific issue is where the definition of noninline member functions and static data members must be specified.

There are at least three different models that are either in use or have been discussed:

1. All definitions must be present in all compilations (in order for the automatic instantiation mechanism to be guaranteed to work). As I understand it, this is the approach used by the Borland compiler.

   The advantage to this approach is that it allows users to organize their source code (e.g., file names, directories, etc.) in any form desired as long as the necessary text is included at compile time. Another advantage is the fact that it makes dependencies between template definition files easy to see (e.g., tools like makedepend will work automatically, even when using templates).

   The disadvantage is that it may result in much larger compilations than are really required. Another disadvantage is the fact that some processors may not want to process the definitions at this point in time (e.g., cfront-style link time instantiators).

2. The definitions may be found in a file name "related" to the file name in which the template declaration was found (e.g., the definitions for templates declared in List.h would be found in List.c). This is the approach used by cfront and others seeking to be compatible with cfront.

   The disadvantages of this approach is that it enforces a particular set of naming conventions on the user, and that it makes dependencies more difficult to understand.

3. The definitions are placed in an arbitrary .c file that is separately compiled. The output of the compiler is used at some other unspecified time along with the output of compilations that make use of the template to generate the necessary instantiations. I'm not aware of any compilers that make use of this model.

I believe that if the WP is to mandate a compilation model, it must be capable of supporting the most commonly used compilation models with as little change to existing source code as possible.

I propose that a new preprocessing directive be added to the language. The new directive would use a syntax similar to the `#include` preprocessing directive.

```
#templatedefs "filename"
#templatedefs <filename>
```

This directive would be placed in header files that declare templates and would provide the name of the file in which the template definitions for noninline functions and static data members could be found.

I think the new command needs to be a preprocessing command so that model #1 above can be supported. In model #1, the new directive would essentially be treated as an `#include`.

The preprocessing level also seems right because the preprocessor already includes the concept of a source file name (in `#include` directives), while the actual language lacks such a concept. Furthermore, the file name should go through the same search path processing that normal include file names go through. This restricts processing of file names to the preprocessing phase of translation and the higher level language can remain ignorant of such things.

For compilers that implement model #2, the new directive would essentially be ignored by the preprocessor (or perhaps translated into something else, like a pragma) and the file name would be passed through to the compiler. The compiler would then record the

file name along with the other information saved about templates and would then use the specified file name at whatever point it was needed to generate instantiations.

This directive allows complete freedom for an implementation to choose when to process the template definition file. It permits the definition file to be processed during all compilations, selected compilations (at the compiler's discretion), or at some point after compilation (such as a link time mechanism).

If the specified file is processed at some point, it will be processed as if it were textually included at an unspecified point following the `#templatedefs` directive in one of the files in which the `#templatedefs` directive appeared.

The Sun[3] and EDG compilers both currently make use of a mechanism (in certain compilation modes) in which a template definition file is included at some point during the compilation process[4]. User code should not rely on inclusion at a particular point, so it seems best to leave the inclusion point as "unspecified". The Sun and EDG compilers do, however, rely on different mechanisms to generate the name of the file to be included.

The prospect of adding a preprocessing directive is admittedly a difficult one to swallow, but I think a necessary one. C++ already requires a different preprocessor to accommodate `//` style comments. The additional processing required to handle the `#templatedefs` directive would be trivial (but would be compiler dependent).

I've gotten initial comments on this proposal from a few people. I would summarize their comments as follows:

- A new preprocessing directive is ugly and is likely to be unpopular among members of the committee.
- However, preprocessing is probably the right place for this information to be provided.

Status: Open

Version added: 7
Version updated: 7

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 {
```

[3]This is based on my (hopefully correct) understanding of how Sun handles instantiations. I hope someone from Sun will correct me if I have misunderstood.

[4]Actually, both compilers currently do the inclusion at the end of the primary source file

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

class C {};
template <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::f;  // May be placed here
        void foo(){
                template T::f;  // 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