X3J16/94-0020
WG21/N0407

# Template Issues and Proposed Resolutions
**Revision 5**

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

January 25, 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.

## 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 fine 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

## 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?

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?

2.9 The previous rule makes possible runaway recursive instantiations. How should an implementation prevent this?

2.10 At what point are names injected?

2.11 Does an array parameter decay to a pointer type?

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?

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

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?

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?

2.18 When must a type used within a template be completed?

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

2.20 Proposal to defer error checking for `operator ->`.

2.21 When are names considered known in a template dependent base class?

## 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?

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?

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.

3.10 What happens when the explicit specification of function template arguments results in an invalid type?

3.11 How do default arguments work when using new explicit specialization declarations?

3.12 How do old style specialization declarations interact with new style ones?

3.13 Revisiting default arguments.

## 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?

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?

## 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++.

6.9 Should there be a translation limit that specifies a minimum depth of recursive instantiation that must be supported?

6.10 Can a single template declaration declare more than one thing?

6.11 Can a storage class be specified in a template parameter declaration?

6.12 Can an incomplete type be used as a template argument?

6.13 Can a template nontype parameter have a void type?

6.14 Can a nontype parameter be a floating point type?

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?

### 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

2.6 Question: Can any conversions be done on nontype actual arguments of class templates?

Answer: Yes.

Trivial conversions, promotions, and standard conversions should be allowed. The following are examples of allowed conversions:

```
template <long I> struct A {};

const  short si = 1;
const  char ci = 1;
const  int ii = 1;

A<si>  asi;     // short -> long
A<ci>  aci;     // char -> long
A<ii>  aii;     // int -> long

template <char* C> struct B {};

char    c[10];
B<c>    bc;     //  array to pointer
B<0>    bnull;  //  0 to null pointer

struct  Base {};
struct  Derived : public Base {};

template <Base& B> struct C {};

Derived  d;
C<d>    cd;     //  Derived to base

template <int Derived::* dpm> struct D {};

D<&Base::i>  dbi;       // Base to derived pointer-to-member

template <const int I> struct E {};
E<1> e;         // int -> const int
```

Remarks: The current rule that no conversions are allowed is considered to be too restrictive. Any proposals to allow overloading of class templates would need to take these conversions into account.

Status: Open

Version added: 1
Version updated: 1

2.8 Question: How can a class template name be used within the definition of the template?

Is it permitted for a class template to refer to an instance of itself? For example:

```
template <class T> class A {
        struct B {};
        A*            p1;   // OK
        A<T>*         p2;   // OK
        A<T*>*        p3;   // OK (infinite instantiation without
                            //     previous rule)
        A<int>*       p4;   // OK

        int A::*      pm1;  // OK
        int A<T>::*   pm2;  // OK
        int A<T*>::*  pm3;  // OK (infinite instantiation without
                            //     previous rule)
        int A<int>::* pm4;  // OK

        A             mem1; // Error - incomplete type
        A<T>          mem2; // Error - incomplete type
        A<T*>         mem3; // OK (possible infinite instantiation)
        A<int>        mem4; // Error unless specific definition
                            // exists

        A::B*         pb1;  // OK
        A<T>::B*      pb2;  // OK
        typedef A<T*>::B;
        A<T*>::B*     pb3;  // OK - but requires designation as type
        typedef A<int>::B;
        A<int>::B*    pb4;  // OK - but requires designation as type
};
```

`A` and `A<T>` refer to the class template and may be used anywhere a class name can normally be used within its class definition. `A<T*>` and `A<int>` cannot, because they may refer to specific definitions of `A`.

`A<T*>` refers to a nonreal instantiation (an instantiation of a template with an argument that contains a template parameter type). Nonreal instantiations can be used as completely defined object types in the scanning of a class template but they cannot be used in a context that requires any knowledge of the type's properties because it is not possible to know which definition of `A` will be used (i.e., a specific definition could be supplied later). Consequently, use of names such as `A<T*>::B` must be explicitly designated as types (this requirement may be revised when specialization issues are reviewed).

`A<int>` refers to a instance of the template (either generated or a specific definition). If `A<int>` refers to a generated instance it will be treated as a nonreal instantiation like `A<T*>`

(because the class template is not yet complete and so the instance cannot be generated in order to inspect its properties), which means that references such as `A<int>::B` must be explicitly designated as a type. If `A<int>` refers to a specific definition supplied before the template definition then it may be used in the class definition with no restrictions. For example,

```
template <class T> class A;

class A<int> {};

template <class T> class A {
        A<int>      a;      // OK
};
```

This rule is also derived from 92-0133/N0209. Until the complete class template has been defined, the only instantiations that are possible are instantiations of incompletely defined object types (because the instantiation is considered to take place immediately following the class template definition). Consequently, generated instances may only be used within the class template in contexts that permit incomplete object types.

Status: Tentatively approved by the extensions working group in Munich. To be revisited when specialization issues are reviewed.

Version added: 1
Version updated: 3

2.9 Question: The previous rule makes possible runaway recursive instantiations. How should an implementation prevent this?

Answer: The result of an infinite recursion is undefined.

Remarks:

A new issue will be added regarding a translation limit for the maximum recursion depth.

The ability to terminate a recursive instantiation at some point is needed to prevent simple program errors from causing infinite instantiations that might otherwise result in compiler crashes. The following example illustrates how a misplaced "*" can result in a recursive instantiation.

```
template <class T> struct A {
        A<T*>   a1;  // Oops - recursive instantiation
        A<T>*   a2;  // This is what was intended
};

A<int>        a;
```

Status: Approved in San Jose.

Version added: 1
Version updated: 1

2.10 Question: At what point are names injected?

Answer: It is an error (ill formed) to reference a class or function in a friend declaration in a class template that has not been declared prior to the instantiation that references

the class or function. Likewise, it is an error to reference a name in an elaborated type specifier in a function parameter type or return type that has not been declared prior to the instantiation that references the name.

Note that friends may still be defined in class templates. They must simply have already been previously declared.

```
template <class T> struct A {
        friend void f(A<T>){}
        friend void g(A<T>);
        friend void f2(struct X* x); // Error during instantiation
                                     // f2 undefined
        void mem_func(struct Y* y);  // Error during instantiation
                                     // Y undefined
        friend class B; // Error during instantiation
                        // B undefined
        friend class C<T>;
};

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

void main()
{
        void* fp;
        X* x;
        A<int> a;
        A<char> ac; // Error during instantiation
                    // f(A<char>) undefined
}
```

Note that there are two kinds of injection: friend injection (`f`, `g`, `f2`, `B`, and `C` in the example above) and injection of class names used in function declarations (`X` and `Y` in the example above). These are handled consistently.

The familiar complex example would be written as follows:

```
// f() is not dependent on the template parameter so
// its declaration must precede the template definition.
void f();

template <class T> struct Complex {
        friend Complex operator +(Complex, Complex) { /* ... */ }
        friend void f();
};

// operator + is dependent on the template parameter so its
// declaration may follow the template definition.
```

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

Complex<float> cf;
```

In the following example

```
template <class T> struct A {
        friend void f(T);
};
int f = 0;
```

if no instantiations were ever created, should an error be issued?

This is covered by issue 6.2 which says that errors in templates may be issued when the template declaration is scanned but may be deferred until an instantiation is generated. In other words, a diagnostic may be issued but is not required.

Status: Open

Version added: 5
Version updated: 5

2.11 Question: Does an array parameter decay to a pointer type?

```
template <int a[5]> struct S {};
int *p;
S<p> x;  // Error
```

Answer: No. The decay of arrays to pointers is largely a carryover from C for compatibility. There is no compatibility issue when using template parameters so this is not needed.

Status: Open

Version added: 1
Version updated: 1

2.14 Question: Can a class template or function template be declared as a friend of a class?

Answer: Yes. In this example all instances, both generated and user supplied specific definitions, of class template `B` and of `void f(T)` are friends of class `A`. Class `A` may itself be a class template. A template may first be declared by a friend declaration in a class or in a class template. A template may not be defined in a friend declaration.

```
class A {
        template <class T> friend class B;
        template <class T> friend void f(T);
        template <class T> friend class C {};  // Error
};
```

Status: Tentatively approved by the extensions working group in Munich (access control issues to be revisited when specialization is reviewed). I believe this was inadvertently omitted from the issues voted on in San Jose and will need to be voted on at a subsequent meeting.

Version added: 2
Version updated: 3

2.16 Question: 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?

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

template <class B> void A<B>::f()
{
        B* b = 0;  // Template parameter B or base class B?
}

A<int> a;
```

We have disallowed the redefinition of a template parameter, but this case is not really a redefinition.

Answer: The member is preferred over the template parameter in this case.

Status: Open

Version added: 5
Version updated: 5

2.17 Question: 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?

```
struct A {
        struct B {};
};

template <class B> struct C : public A {
        B        b;  // Template parameter B or base class B?
};

C<int>        c;
```

Answer: The base class member is used in preference to the template parameter.

Status: Open

Version added: 5
Version updated: 5

2.18 Question: When must a type used within a template be completed?

```
class A;
template <class T> void f(T)
{
        A a;  // Error
}

class A {};
```

Answer: The name binding rules that have been passed specify that `A` is bound when it is seen in the template definition. If a name is used in a context that requires a complete type, then it must be complete at the binding point.

As is the case with other errors, this error may be diagnosed when the template definition is seen but is not required to be diagnosed until an instantiation of `f()` is performed.

Status: Clarification.

Version added: 5
Version updated: 5

2.19 Question: Must a specialization declaration precede the use of a class template in a context that requires only an incomplete type?

```
template <class T> struct A {};
A<int>* a;
struct A<int> {};
```

Answer: No, a specialization declaration is only required before the first use that requires a complete type.

Status: Clarification.

Version added: 5
Version updated: 5

2.20 Proposal to defer error checking for `operator ->`.

It has been suggested by Dag Brück, Paul Lucas, and possibly others, that the current language definition makes in unnecessarily difficult to write class templates that provide `operator ->` functions, but that can be instantiated for types that are not necessarily class types.

The following changes are proposed to support this usage:

1. For generated instantiations of class templates only, an `operator ->` function is permitted to return an object that is not a class type. It is still an error to declare such a function in a nontemplate class or in a user specialization of a class template.
2. When a class has an `operator ->` function that returns a nonclass type, it is an error to call the function or to cause its address to be taken. Such an `operator ->` function may not be declared virtual.

```
template <class T> struct Ptr {
        T object;
        T* operator ->() { return &object;}  // OK
};

Ptr<int> a;  // OK

int f()
{
        &Ptr<int>::operator->;  // Error - address taken
}
```

```
// Nontemplate class
struct Xyz {
        int* operator ->();  // Error
};

// User specialization of a template class
struct Ptr<char> {
        char* operator ->();  // Error
};

// User specialization of a template member function
float* Ptr<float>::operator ->()  // Error
{
        static float f;
        return &f;
}
```

Status: Open

Version added: 5
Version updated: 5

2.21 Question: When are names considered known in a template dependent base class?

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

struct A<int> {
        struct C {};
};

template <class T> struct X : public A<T> {
        A<T>::C      c;  // typedef A<T>::C needed
};

struct X<char> : public A<char> {
        A<char>::B   b;  // typedef A<char>::B not needed
};

A<int> ai;
A<char> ac;
```

Answer: A name from a template parameter dependent base class cannot be known to be a type or nontype at the time the template definition is scanned, so such types must be explicitly declared to be types.

In the example above `A<char>` is not dependent on a template parameter, so `A<char>::B` is known to be a type and needs no explicit designation.

Status: Clarification

Version added: 5
Version updated: 5

## Function Templates

3.2 Question: Can the parameters with default arguments involve template parameters in their types?

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

Answer: This issue has been resolved by the new rules concerning explicit qualification of function template calls.

Status: Closed

Version added: 1
Version updated: 5

3.4 Question: Can any conversions be done when matching arguments to function templates?

Answer: This issue is now part of issue 3.9.

Status: Closed

Version added: 1
Version updated: 5

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

```
template<class T> void f(T::A 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
}
```

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.

Status: Open

Version added: 1
Version updated: 5

3.9 A proposal to allow conversions in function template calls.

This proposal recommends that the overload resolution rules be modified to allow certain conversions to be performed on the function arguments.

1. For function arguments whose type does not depend on a template parameter the full set of argument conversions permitted for nontemplate functions should be allowed.
2. For function arguments whose type does depend on a template parameter the only conversion considered is a conversion from a derived class to a base class (i.e., from `Derived<T>` to `Base<T>`).

These conversions are possible because, when applied using the rules described below, neither affects the types that will be deduced to generate an instantiation.

This example illustrates the proposed behavior for conversions applied to arguments that do not depend on template parameters:

```
template<class T> void f(T*, int);     // f1
template<class T> void f(T, char);     // f2
void f(char,char);                     // f3
void g(int* p, char c, int i)
{
        f(p,i);  // f<int>(p,i)        // f1
        f(p,c);  // f<int*>(p,c)       // f2
        f(i,i);  // f<int>(i,char(i))  // f2
        f(c,c);  // f(char,char)       // f3, prefer
                                       //     non-template
        f(i,c);  // f<int>(i,c)        // f2
}
```

This example attempts to implicitly convert a `D<T>` to a `B<T>`:

```
template <class T> struct B { };
template <class T> struct D : public B<T> {};
template <class T> void f(B<T>&) {}
```

```
void main() {
        B<int>    b;
        D<int>    d;

        func(b);
        func(d);  // OK? -- requires D<T> -> B<T> conversion
}
```

This is prohibited by the current WP but is required to support polymorphic template functions. The importance of this feature is demonstrated by the fact that is has been implemented by many compilers. Of the compilers to which I have access that support templates, three of four implement this feature (of course no two implement it in exactly the same way).

Allowing conversions requires that the overload resolution rules for template functions and other functions of the same name (WP 14.4) be revised. Recall that the current rules are:

1. Look for an exact match (WP 13.2) on functions; if found, call it.
2. Look for a function template from which a function that can be called with an exact match can be generated; if found call it.
3. Try ordinary overloading resolution (WP 13.2) for the functions; if a function is found, call it.

The current rules for overload resolution require that arguments to template functions exactly match the corresponding parameter types. Not even trivial conversions are allowed. Most implementations have decided that the restriction on trivial conversions is too strict and have extended the search for a matching template function to include trivial conversions. The existing rules can be adapted to include trivial conversions of template function arguments with acceptable results.

In contrast, adding the full set of conversions for arguments that don't depend on template parameters and adding the `Derived<T>` to `Base<T>` conversion require, for the first time, that template functions and nontemplate functions be considered side by side for overload resolution purposes. Rule #3 must be modified to consider both template functions and nontemplate functions where it previously was only required to deal with nontemplate functions.

The proposal is to eliminate the existing rules described in WP 14.4 and to extend the general overload resolution in WP 13.2 to handle template functions. The new rules, in essence, include template functions in the normal overload resolution process (although they allow only a subset of the normal conversion operations) and use the fact that a function is or is not a template function as a tie-breaker to prefer nontemplate functions if all other conditions are equal.

In more precise terms, the proposed overload resolution algorithm is:

1. Determine how well the actual arguments match each function. For template functions, each argument match is rated as usual but no matches other than exact matches and standard conversions involving a conversion from a derived class to a base class are considered for arguments whose type depends on a template parameter.

2. Find the intersection of the sets of functions that best match on each argument.
3. Added step: If the best-match set contains both template functions and nontemplate functions, eliminate the function templates. That is, all other things being equal, a function template is considered a worse match than a nontemplate function.
4. If the best-match set contains more than one function, the call is ambiguous. If it contains no functions, the call is illegal. If it contains exactly one function, the function must be a strictly better match for at least one argument than every other possible function (but not necessarily the same argument for each function). Otherwise, the call is illegal.

The following examples may be helpful to illustrate why the rules are required and how they would work:

```
struct B { };
struct D : public B { };
struct F : public D { };
template <class T> void f(T, B*) {}
template <class T> void g(T, D*) {}
void f(int, D*);
void g(int, B*);
void m () {
        F *p = new F;
        f(0, p);  // Nontemplate f chosen
        g(0, p);  // Template g chosen
}
```

The nontemplate `f` should be chosen because on the second argument (which does not involve a template parameter type) the template `f` requires a standard conversion (`F*` ⇒ `B*`) that is less desirable than the one for the nontemplate `f` (`F*` ⇒ `D*`). Likewise, the template `g` should be chosen because on the second argument the nontemplate `g` requires a standard conversion (`F*` ⇒ `B*`) that is less desirable than the one for the template `g` (`F*` ⇒ `D*`).

```
template <class T> struct B { };
template <class T> struct D : public B<T> {};
struct F : public D<int> { };
template <class T> void f(B<T>&) {}
void f(D<int>&) {}
template <class T> void g(D<T>&) {}
void g(B<int>&) {}

void m() {
        F x;
        f(x);  // Nontemplate f chosen
        g(x);  // Template g chosen
}
```

The nontemplate `f` should be chosen because on the argument (which does involve a template parameter type) the template `f` requires a standard conversion (`F*` ⇒ `B<int>*`)

that is less desirable than the one for the nontemplate f (`F*` ⇒ `D<int>*`). Likewise, the template `g` should be chosen because nontemplate `g` requires a standard conversion (`F*` ⇒ `B<int>*`) that is less desirable than the one for the template `g` (`F*` ⇒ `D<int>*`).

The proposed rules do introduce a case that may be handled differently by the new rules than it is by existing implementations. I say "may" because it depends on whether the implementation extends exact matches of template functions to include trivial conversions. In the following example existing implementations diagnose this call of `f` as ambiguous. Under the new rules the algorithm would consider the conversion from `int` to `const int&` to be one of the "less desirable" exact matches and would therefore select the other template. This is the only case known where the generalized algorithm would yield a different result than the WP algorithm extended to allow trivial conversions, but it seems that even this change is desirable because it makes the template overloading rules more consistent with the nontemplate rules.

```
template <class T> void f(T) {}
template <class T> void f(const T&) {}
int main()
{
        int i;
        f(i);
}
```

Status: Open

Version added: 2
Version updated: 5

3.10 Question: What happens when the explicit specification of function template arguments results in an invalid type?

This issue concerns the interaction of two features: the ability to have a template parameter whose type depends on another template parameter, and the explicit specification of function template arguments.

When the type of one template parameter depends on another template parameter it is possible for an invalid type to be produced. For example

```
template <class T, T* t_ptr> struct A {};

A<int, 0> a1;   // OK
A<int&, 0> a2;  // Error - cannot have a pointer to a reference
```

It is now possible for this same problem to occur with function templates. The question is, how should this kind of error interact with overload resolution?

Consider this example:

```
template <class T> struct A {
        T* T_ptr;
        typedef int X;
};
```

```
template <class T> struct B {
        T t;
        typedef float Y;
};

template <class T, A<T>::X f, class T2> T f(T2){}  // #1
template <class T, B<T>::Y f, class T2> T f(T2){}  // #2

int main()
{
        int i;
        f<int, 1>(i);   // Presumably intended to call #1
                        // because second an int
        f<int&, 1>(i);   // Presumably intended to call #1
                         // because second an int
}
```

In the second call of `f`, the template parameter `T` is `int&`. Note that it is not possible to instantiate `A<int&>` because `T_ptr` would have a type of "pointer to reference" which is not allowed.

The problem is that you have to instantiate `A<int&>` to determine that there is a problem. I believe that this case needs to be an error. The alternative would be that the compiler needs to determine whether the instantiation of `B<T>` resulted in an error and, if so, pretend that the instantiation was not done. I think this would be a mistake.

Answer: I would suggest a rule that says something like "if you supply a given template argument that argument will be compared with every possible function template in the list of overloaded functions. If use of the template argument in a given function gives rise to an ill-formed construct then the program is ill-formed."

Status: Open

Version added: 5
Version updated: 5

3.11 Question: How do default arguments work when using new explicit specialization declarations?

```
template <class T> void f(T, T, int = 1){}
void f(int, int, int){}  // Old style specialization
void f<>(int, int, int){}  // New style specialization (syntax?)
```

The old style specialization did two things: it provided the body for a particular instance of the template, and it provided a function declaration that was treated as a normal (nontemplate) function for overload resolution purposes.

The new style specialization does only the former. It does not affect overload resolution. Consequently, it must not be able to do anything that would affect the way in which the function is handled by overload resolution. For example, it must not be able to have default argument characteristics that are different than those specified by the template.

It could be argued that the specialization must only match the template in terms of which parameters have default arguments but not in the value of the default arguments. This is problematic because of the ability to add default arguments in subsequent declarations of the template.

Answer: Default arguments may not be specified in the declaration and/or definition of specializations.

Status: Open

Version added: 5
Version updated: 5

3.12 Question: How do old style specialization declarations interact with new style ones?

```
template <class T> void f(T){}
void f(int){}  // Old style specialization
void f<>(int){}  // New style specialization (syntax?)
```

Answer: The old style declarations, as always, affect overload resolution but say nothing about whether the body of the function will be generated based on the template or will be supplied by a specialization. In the absence of a new specialization declaration the body will be generated from the template. If a body is to be supplied by a specialization and the old style overload resolution is desired then both the old style and new style declarations must be supplied.

An old style declaration may include a function body, in which case it is also considered to be a specialization declaration. Such a definition must precede any calls of the function in the translation unit.

Status: Open

Version added: 5
Version updated: 5

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?
}
```

Status: Open

Version added: 5
Version updated: 5

## Member Function Templates

4.4 Question: What are the rules regarding use of the inline keyword in member function declarations?

Answer: If a member function is declared inline in the class template then the template definition and any specific definitions will also be inline (even if the `inline` keyword is not present in the template definition or specific definition). If the member function is not declared inline in the class template then the template definition and any specific definitions may or may not be declared inline. This is simply an extension of the current rules for inline functions of classes extended to address class templates.

It should be noted that if the template definition is declared inline outside of the class template and a noninline specific definition is provided, the noninline specific definition cannot be called from a file that includes the inline template definition. In example B, there is no way to indicate that a specific definition of `A<int>::f` has been supplied in another file and that the inline version should not be generated.

example A:

```
template <class T> struct A {
  void f();
};
A<int> a;
template <class T> inline void A<T>::f(){}
void A<int>::f(){}  // Not inline
```

example B:

```
template <class T> struct A {
  void f();
};
template <class T> inline void A<T>::f(){}
void x()
{
        A<int> a;
        a.f();  // Calls template version not specific definition
}
```

Status: Open

Version added: 1
Version updated: 1

4.6 Question: Can a class template member function be redeclared outside of the class?

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

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

Answer: No. This kind of redeclaration is not allowed for normal classes nor is it allowed for class templates.

Status: Clarification

Version added: 5
Version updated: 5

## Other Issues

6.8 Question: May template declarations be given a linkage specification other than C++.

```
extern "C" {
        template <class T> void f(){}   // Error
        template <class T> struct A {   // OK
                void f();
        };
}
```

Answer: Function templates must have C++ linkage. Class templates may be given other kinds of linkage.

Status: Open

Version added: 5
Version updated: 5

6.9 Question: Should there be a translation limit that specifies a minimum depth of recursive instantiation that must be supported?

```
template <unsigned int I> struct A {
        A<I+1> a;
};

struct A<10U> {};

A<0U> a;   // OK - completes after 10 instantiations
A<11U> a2; // ? - completes after 0xffffffff iterations
```

Answer: Recursive instantiations are rare in real use but can easily occur by accident. It is a significant convenience to the user if the compiler can terminate the recursion in an orderly fashion (e.g., by issuing an error message) instead of having the compiler abort when some resource (such as memory or disk space) is exhausted.

A translation limit will give programmers guidance as to the recursion depth that they can expect an implementation to provide while allowing implementations to reasonably catch runaway recursions.

Real (i.e., not contrived) examples of recursion seem to be very shallow so a small limit (somewhere in the range of 10 or 20) is probably reasonable.

Note that any limit that may be chosen does not require a given implementation to be able to instantiate N recursive instantiations if those instantiations result in some other translation limit being exceeded.

Status: Open

Version added: 5
Version updated: 5

6.10 Question: Can a single template declaration declare more than one thing?

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

Answer: No

Status: Clarification

Version added: 5
Version updated: 5

6.11 Question: Can a storage class be specified in a template parameter declaration?

```
template <register int I> struct A {};  // Error
```

Answer: No

Status: Open

Version added: 5
Version updated: 5

6.12 Question: Can an incomplete type be used as a template argument?

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

Answer: Yes, provided the template parameter is only used within the template in contexts that do not require a complete type. If the parameter is used in a context that requires a complete type then the type must be complete at the point at which the instantiation of a complete type is performed.

```
template <class T> struct A {
        T t;
};
struct B;
A<B>* ap;  // OK complete instantiation of A<B> not required
A<B>  a;   // Error - complete instantiation of A<B> needed
struct B {};
A<B>  a2;  // OK - B is now complete
```

Status: Open

Version added: 5
Version updated: 5

6.13 Question: Can a template nontype parameter have a void type?

```
template <void V> struct A {};      // Error
```

Answer: No. There is no way to supply a void value with which to generate an instantiation.

Note: `void *` is different from `void` and is unrelated to this issue.

Status: Open

Version added: 5
Version updated: 5

6.14 Question: Can a nontype parameter be a floating point type?

```
template <double d> struct A {};  // Error
template <double& d> struct B {};  // OK
template <double* d> struct C {};  // OK
```

Answer: No

Note: This could be supported, but would require adding a "floating point constant expression" that currently does not exist in the language.

Status: Open

Version added: 5
Version updated: 5

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

Answer: For parameters of arithmetic types the argument may be a constant expression of the appropriate type including casts from one arithmetic type to another. For parameters of pointer and reference types the argument must be the address of an entity with external linkage. No operators or casts are allowed.

```
template <int I> struct A {};
template <int* I> struct B {};
template <double D> struct C {};  // Depends on issue 6.14

A<1+1>           a1; // OK
A<(int)2.0>      a2; // OK

int i;
char c;
int iarray[10];

struct X {
        int i;
        static int si;
};
X x;

B<&i>            b1; // OK
B<iarray>        b2; // OK
B<&iarray[0]>    b3; // Error - array references not allowed
B<&iarray + 1>   b4; // Error - address arithmetic not allowed
B<(int *)&c>     b5; // Error - nonarithmetic casts not allowed
B<&x.i>          b6; // Error - field selection not allowed
B<&X::si>        b7; // OK

C<1.0 + 2.0>     c1; // Depends on issue 6.14
C<(double)1>     c2; // Depends on issue 6.14
```

Status: Open

Version added: 5
Version updated: 5

6.16 Question: Can a template parameter be used in an explicit destructor call?

```
template <class T> inline void destroy(const T& t)
{
        T* t2 = new T;
        t.~T();         // OK
        t2->T::~T();    // OK
}

struct A {};

struct B {
        ~B(){}
};
```

```
int main()
{
        A a;
        B b;
        destroy(a); // Class with no destructor - OK
        destroy(1); // Nonclass type - OK
        destroy(b); // Class with destructor - OK
}
```

Answer: Yes

Status: Open

Version added: 5
Version updated: 5