X3J16/93-0074
WG21/N0281

Template Issues and Proposed Resolutions

Revision 2

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

May 28, 1993

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.

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 and only touches on some of the more global issues such as name binding, and avoids altogether issues such as managing function instantiations across separate compilation units. Many of these issues are discussed in Bjarne Stroustrup's paper "Major Template Issues" (93-0081/N0288).

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.

Template Parameters

1.1 Can template parameters have default arguments?

1.2 Where can default arguments for template parameters be specified?

1.3 Can a type parameter be used in the type declaration of a nontype parameter?

1.4 Can a nontype parameter as used above have a default argument?

1.5 Should it be possible to redeclare a template parameter name to mean something else inside a template definition?

1.6 Can the name of a nontype parameter be omitted?

1.7 Can the name of a type parameter be omitted?

1.8 Can a typedef appear in a template declaration?

1.9 Can a nontype parameter have a reference type?

1.10 Are qualifiers allowed on nontype parameters?

1.11 May a template parameter have the same name as the class template with which it is associated?

Class Template References

2.1 Can a nontype parameter that is not a reference be used as an lvalue or have its address taken?

2.2 Can the class template name be used as a synonym for the current instantiation inside a class template?

2.3 Can a class template have a template parameter as a base class?

2.4 Can a local type be used as a type argument of a class template?

2.5 Can a character string be a nontype argument?

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

2.7 What causes a template class to be instantiated?

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?

2.13 Can template parameters be used in elaborated type specifiers?

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?

Function Templates

3.1 Can function templates have default function parameters?

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?

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?

3.6 Can unnamed types be used as template arguments?

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?

3.9 A proposal to allow `Derived<T>` to `Base<T>` conversions in function template calls.

Member Function Templates

4.1 Are inline member functions that are not used by a given class template instance instantiated?

4.2 Can a noninline member function or a static data member be instantiated when referenced?

4.3 Must the template parameter names in a member function definition match the names used in the class definition?

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?

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?

5.2 Can you declare an incompletely defined object type that is a specific definition of a class template?

5.3 Can the class template name be used as a synonym for the current specific definition inside the specific definition?

5.4 Can a specific definition of a class template be a local class?

Other Issues

6.1 Should classes used as template arguments have external linkage?

6.2 When must errors in template definitions be issued and when must they not be issued?

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

6.4 Can a static data member of a class template be declared with an incomplete array type?

6.5 How should template arguments that contain "`>`" be parsed?

6.6 Can template versions of `operator new` and `operator delete` be declared?

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?

Nontype Parameters for Function Templates

A proposal for nontype parameters for function templates as required by the `Bitset` class.

Template Parameters

1.1 Question: Can template parameters have default arguments?

```
template <class T, int I = 1> struct A {};      // OK
template <class T = int> struct B {};           // OK
```

Answer: Defaults are allowed for parameters of class templates but not function templates.

Rationale: Defaults are allowed for nontype parameters based on the current syntax. Defaults are not needed for function templates because the template arguments of function templates are deduced from the types of the arguments with which the function is called.

Status: Support for default arguments for nontype parameters was tentatively approved by the extensions working group in Portland. Subsequently objections were raised regarding the omission of defaults for type parameters. The proposed resolution has been modified to include type parameters. This issue will be revisited by the extensions working group.

Version added: 1
Version updated: 2

1.2 Question: Where can default arguments for template parameters be specified?

```
template <int i, int j, int k = 1> class A;             // OK
template <int i, int j = 1, int k> class A {            // OK
        void f();
};

template <int i = 1, int j, int k> void A<i>::f(){}     // OK
```

Answer: Default arguments for template parameters may be specified in a class template declaration, definition, and on the definition of a member functions and static data members that are outside of the class definition.

After merging the default argument information from the various possible sources, the last parameter with a default argument may not be followed by a parameter that has no default argument.

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.3 Question: Can a type parameter be used in the type declaration of a nontype parameter?

Answer: Yes.

```
template <class T, T t> class A {
public:
        T       a;
        A(T  init_val = t) { a = init_val; }
};

template <class T, T (*fp)(T)> struct B {
        T       t;
        B()  { (*fp)(t); }
};
```

Remarks: This introduces some cases where type checking for certain parts of a function template declaration must be deferred until a type is created for an instance of the template. For example,

```
template <class T, T t> class A {};
template <class T> A<T, 10> f(T);  // 10 cannot be type-checked yet
```

There are several different types of instantiations that can be done depending on whether the template actual arguments are "normal" types and constants or whether they are the template parameters of another template declaration. For purposes of this discussion let us define an instantiation in which none of the arguments are template parameters as a "real" instantiation (e.g., `A<int, 1>`) and an instantiation in which any argument is a template parameter as a "nonreal" instantiation (e.g., `A<T,I>`). In a nonreal instantiation a nontype argument whose type is a template parameter cannot be type checked until the template associated with the parameter (function template `f(T)` in the example above) is instantiated.

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.4 Question: Can a nontype parameter as used above have a default argument?

Answer: Yes – for example, this should be legal:

```
template <class T> T f(T){}
int  xyz(int){}
template <class T, T (*fp)(T) = f> struct C {
        T       t;
        C()  { (*fp)(t); }
};

C<int>  c;
C<int,  xyz> c2;
```

Remarks: This is similar to the previous example; type checking for the default argument cannot be done until a real instantiation of class template `C` is generated.

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.5 Question: Should it be possible to redeclare a template parameter name to mean something else inside a template definition?

Answer: Redeclaration of a template parameter name anywhere within the scope of a class template, function template, or member function of a class template is prohibited. Redeclaration in a scope within one of these scopes is also prohibited.

Rationale: The redeclaration rules should be consistent so that a given usage is either always permitted or always prohibited. The following four cases should be handled similarly:

1. Class templates and classes nested within class templates should follow the same rules. It is now possible to define a nested class outside of the enclosing class and, as with member functions described below, it is important that nested classes declared inside a class declaration be handled the same way as those declared outside. Redeclaration of `T` should be disallowed (or allowed) in both of the following cases:

```
template <class T> class A {
  class B {
    int T;  // Error
  };
};

template <class T> class A {
  class B;
};
template <class T> class A<T>::B {
  int T;  // Error
};
```

2. Member functions defined in the body of a class template should follow the same rules as member functions defined outside of the class template. Redeclaration of `T` should be disallowed (or allowed) in both of the following cases:

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

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

3. Function templates should follow the same rules as member functions of class templates. Redeclaration of `T` should be disallowed (or allowed) in both of the following cases:

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

template <class T> void g(T)
```

```
{
  int T;  // Error
}
```

4. Local classes of function templates and local classes of member functions of class template classes should follow the same rules as nested classes of class templates. Otherwise there is likely to be confusion about the difference in handling between nested classes and local classes.

```
template <class T> class A {
  class B {
    int T;  // Error
  };
};

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

The result of these "meta-rules" is that redeclarations are prohibited everywhere except a block scope within a function template or member function. Even use in this context (block scopes) is likely to be error-prone. So, rather than have a single exception (block scopes) the proposed rule is to prohibit redeclaration of template parameter names anywhere within the scope of a class template, function template, or member function of a class template.

This, of course, has the disadvantage that some "benign" redeclarations are prohibited. For example, inadvertent use of name that is also a template parameter name in a block scope in a macro would now cause an error. There are problems caused by a decision to either allow or disallow redeclarations in nested scope. The proposed resolution provides a simple and consistent rule that was considered to be the better of the alternatives.

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

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

```
    {
      int T;  // Error
    }
    class C {
      int T;  // Error
    };
}
```

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.6 Question: Can the name of a nontype parameter be omitted?

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

Answer: Yes.

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.7 Question: Can the name of a type parameter be omitted?

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

Answer: No

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.8 Question: Can a typedef appear in a template declaration?

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

Answer: No – a typedef is used to create a synonym for a type and a class template name is not a type name.

Status: Open

Version added: 1
Version updated: 1

1.9 Question: Can a nontype parameter have a reference type?

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

Answer: Yes (see next section for information about actual arguments of reference parameters).

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.10 Question: Are qualifiers allowed on nontype parameters?

```
template <const int i, volatile int j> struct A {};
```

Answer: Qualifiers are allowed. Note that because template arguments must be constant values or addresses "top level" qualifiers have no effect. Other qualifiers, such as `const int*` in the following example, are useful.

```
template <const int i> struct A {}      // const has no effect
template <const int* i> struct B {}     // const does have effect
```

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

1.11 Question: May a template parameter have the same name as the class template with which it is associated?

Answer: No.

```
template <class T> class T {};  // Error
template <class T> class T;     // Error
```

Rationale: In a definition this usage would be error prone (because the template parameter name would be used when the user wanted to use the class template name. The same danger does not exist in a declaration without a definition but is prohibited for consistency.

Status: Open

Version added: 2
Version updated: 2

Class Template References

2.1 Question: Can a nontype parameter that is not a reference be used as an lvalue or have its address taken?

Answer: No.

```
template <int I> struct A {
        void f() {I = 1;}              // Error
        void g() {int* p = &I;}        // Error
        void h() { const int& j = I;}  // OK - temporary created
};
int main()
        A<10>   a;
        a.f();
        a.g();
        a.h();
}
```

Remarks: When a reference parameter has its address taken or is used as an lvalue the address used is the address of the object bound to the reference. In the following example `A<x>::f` sets `x` to one while `A<x>::g` assigns the address of `x` to `p`.

```
template <int& I> struct A {
        void f() {I = 1;}
        void g() {int* p = &I;}
};
int x;
int main()
        A<x>   a;
        a.f();
        a.g();
}
```

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

2.2 Question: Can the class template name be used as a synonym for the current instantiation inside a class template?

Answer: Yes – the following two examples are equivalent

Example 1:

```
template <class T> struct B {
        B(){};
        ~B(){};
        int B::* pmi;
        B*      bp;
        B f1(B& b);
        B f2(B& b) {B b2 = b;return b2;}
        struct C {
                int B::C::* pmi;
                B::C*   bp;
                B::C f2(B::C& b) {B::C b2 = b;return b2;}
        };
        friend B g(B& b);
};
template <class T> B<T> B<T>::f1(B& b) {B b2 = b; return b2;}
```

Example 2:

```
template <class T> struct B {
        B<T>(){};  // Allowed - change from WP
        ~B<T>(){}; // Allowed - change from WP
        int B<T>::* pmi;
        B<T>*   bp;
        B<T> f1(B<T>& b);
        B<T> f2(B<T>& b) {B<T> b2 = b;return b2;}
        struct C {
                int B<T>::C::* pmi;
                B<T>::C*        bp;
                B<T>::C f2(B<T>::C& b) {B<T>::C b2 = b;return b2;}
```

```
        };
        friend B<T> g(B<T>& b);
};
template <class T> B<T> B<T>::f1(B<T>& b) {B<T> b2 = b; return b2;}
```

Note that member functions defined in the class body and member functions defined outside of the class are handled equivalently.

The template arguments may not be supplied in the declarator of the class template as illustrated below.

```
template <class T> struct A {};      // OK
template <class T> struct A<T> {};   // Error
```

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

2.3 Question: Can a class template have a template parameter as a base class?

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

Answer: Yes. The actual argument used to instantiate the class must, of course, be a class.

Remarks: See 6.7 for issues regarding explicit designation of type names.

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

2.4 Question: Can a local type be used as a type argument of a class template?

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

void  f()
{
        class  B {};
        A<B>    a;      //  Legal?
}
```

Answer: No. See also 3.3.

Rationale: Class templates and the classes generated from the template are global scope entities and cannot refer to local scope entities.

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

2.5 Question: Can a character string be a nontype argument?

```
template <char* c> struct A {};
A<"hello">      a;
```

Answer: No—a character string does not have external linkage. Furthermore, different instances of the same string in a given compilation unit may or may not refer to the same address.

Remarks: A similar result may be achieved as follows:

```
template <char* c> struct A {};
char hello_string[] = "Hello";
A<hello_string> a1;
```

Status: Approved by the extensions working group in Portland as an editorial issue.

Version added: 1
Version updated: 2

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. Narrowing conversions are not permitted. 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

2.7 Question: What causes a template class to be instantiated?

Answer:

There are two important consequences of how this issue is resolved. First, programmers will want to know what kinds of usage will cause instantiations and which won't. In this example it is clear that definitions are required for `A<int>::i` and `A<int>::f`, but are definitions required for `A<char>::i` and `A<char>::f`? The programmer must supply specific definitions of `i` and `f` for each instance of class template `A` that is instantiated. The programmer will need to be able to predict which instances are required.

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

void  main()
{
        A<int>     ai;
        A<char>*   ac;
}

void  A<int>::f(){}
int  A<int>::i = 1;
```

Second, there are situations which, depending on the resolution of this issue, can result in circular instantiation problems in which a set of mutually dependent class templates exists. This example causes problems for some compilers. Interestingly, for those compilers, this example can be made to work by removing the definition of class template `B`.

```
template <class T> class B;
template <class T> class C;

template <class T> class A {
        B<T>  *b;
};

// Remove this definition and this will work on compilers that
// would otherwise complain of circular template references.
template <class T> class B {
        C<T>  c;
};
```

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

A<int>  a;
```

The resolution of this issue requires that we distinguish between two types of instantiations: instantiations of incompletely defined object types and instantiations of completely defined object types.

To clarify the terms being used, the following example illustrates the declaration of a incompletely defined object type and a completely defined object type as the terms apply to nontemplate classes.

```
struct X;    // Incompletely defined object type
struct Y {}; // Completely defined object type
```

When a template is referenced in a context that allows an incompletely defined object type, and the template has not yet been instantiated, an incompletely defined object type will be instantiated. In other words

```
template <class T> struct A {};
A<char>*  ac;
```

is equivalent to

```
template <class T> struct A;
A<char>*  ac;
```

If `ac` is later used in a context that requires a completely defined object type, such as `ac->f()`, a completely defined object type is instantiated. Note that the kind of instantiation that is done (complete or incomplete) is independent of whether or not a complete definition of the template exists. If only an incomplete instantiation is required then only an incomplete instantiation will be done even if a full instantiation could be done. If a full instantiation is required and only an incomplete definition of the template has been declared then an error will be issued.

Assuming that the standard defines where completely defined object types and incompletely defined object types can be used, this allows programmers to predict whether or not template definitions need to be supplied for certain types.

This also solves the circular instantiation problem because, in the example above, `B<T>* b` no longer requires an instantiation of a completely defined object type. This has the additional benefit that the behavior of the example is no longer affected by the removal of the definition of class template `B`.

There is one instance in which a completely defined object type is not required but an instantiation of a completely defined object type must still be done if possible. When a pointer to a derived class is used as a function argument it may be necessary to instantiate a completely defined object type for overload resolution to work properly (a derived to base conversion may be required and a completely defined object type is needed to determine the base classes of a type).

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

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

        A::*        pm1;  // OK
        A<T>::*     pm2;  // OK
        A<T*>::*    pm3;  // OK (infinite instantiation without
                          //     previous rule)
        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
        A<T*>::B*   pb3;  // Error - we don't know if A<T*> refers
                          // to a generated instance or
                          // a specific definition
        A<int>::B*  pb4;  // Error unless specific definition
                          // exists
};
```

`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. This is significant because, using the definitions of "free" and "bound" symbols from 92-0133/N0209, `A` is a free symbol as is `A<T>` even though it may appear to be a bound symbol. `A<T>` always refers to an instance of the class template and consequently may be used in ways that the bound symbols `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). `A<int>` refers to a instance of the template (either generated or a specific definition). If `A<int>` refers to a generated instance, it may only be used as an incompletely defined object type within the template definition. 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: Open

Version added: 1
Version updated: 1

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

Answer: The standard should specify a that an implementation may issue an error when a certain recursion depth has been reached. A programmer could then be assured that a program that employed recursive instantiation (but not infinite) would be standard conforming (whatever we decide that means) as long as it doesn't make use of more than `N` levels of recursive instantiation.

Remarks: I don't know whether or not their are legitimate uses for recursive template class references. But it is possible to write a recursive reference that does terminate at some point, such as the following example.

```
template <int I> struct A {
A<I+1> a;
};
struct A<10> {};
A<0> a;
```

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: Open

Version added: 1
Version updated: 1

2.10 Question: At what point are names injected?

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

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

Answer: 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`.

Status: Open

Version added: 1
Version updated: 1

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

2.12 Question: What can be used as an actual argument for a parameter that is a reference?

Answer: A reference parameter references the address of the actual argument. The actual argument may only be an object or function with external linkage as already defined in WP 14.2. A temporary cannot be generated for a reference argument.

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

int x;
static int y;

A<x>    a1;    // OK
A<1>    a2;    // Error - temporary required
A<y>    a3;    // Error - y does not have external linkage
```

Status: Open

Version added: 1
Version updated: 1

2.13 Question: Can template parameters be used in elaborated type specifiers?

Answer: Yes. This is needed to support usage such as:

```
template <class T> class A1 {
        friend class T;
};
template <class T> class A2 {
        friend union T;
};
struct B {};
union C {};

A1<B> a1;    // OK
A2<C> a2;    // OK
A1<C> a3;    // Error - C is not a struct/class
A1<int> a4;  // Error - int is not a struct/class
```

For consistency, template parameter names may be used in elaborated type specifiers in the declarations of class templates, function templates, and member functions of class templates.

Status: Open

Version added: 2
Version updated: 2

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.

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

Status: Open

Version added: 2
Version updated: 2

2.15 Question: Can template arguments be supplied in explicit destructor calls?

Answer: Yes. This example illustrates the ways in which a destructor may be explicitly called. See also 2.2 and 5.3.

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

void main()
{
       A<int>* p;

       p->~A();
       p->A<int>::~A();
       p->A<int>::~A<int>();
}
```

Status: Open

Version added: 2
Version updated: 2

Function Templates

3.1 Question: Can function templates have default function parameters?

Answer: Yes.

```
template <class T> void f(T, int i = 1){}  // OK
```

Status: Open

Version added: 1
Version updated: 1

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: Yes.

If a template parameter is only used in function template parameters that have default arguments there may be instances in which it is not possible to deduce the template parameter types from a given function call. For example,

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

void main()
{
        int i;
        f(i);   // Error - what is type of T?
}
```

The call must supply enough of the function arguments to infer the complete list of template parameters. This could be affected by the proposal to allow explicit qualification of template function arguments.

An alternative would be to require that every template parameter be used in the argument type of at least one function parameter that has no default argument. Using this rule the example above would cause an error would be issued on the declaration of function template `f` because `T` is only used in the type of a function argument with a default argument. This rule would eliminate the possibility of encountering a call in which it is not possible to deduce a template parameter.

Status: Open

Version added: 1
Version updated: 1

3.3 Question: Can a local type be used as a type argument of a template function?

Answer: No. See also 2.4.

```
template <class T> void f(T){}
void main()
{
        class A {};
        A a;
        enum E {e1, e2};
        E e;
        f(a);           // Error
        f(e);           // Error
}
```

Status: Open

Version added: 1
Version updated: 1

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

Answer: Yes. Trivial conversions can be done on all arguments. On arguments that don't involve template types, base to derived conversions can be done.

It would also be possible to allow the complete set of conversions for arguments whose types do not involve template arguments. Should this be allowed?

This subject requires further investigation.

Status: Open

Version added: 1
Version updated: 1

3.5 Question: 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?

Answer: Every template parameter must be used in the argument types of a function template in a context in which, for any given actual argument the resulting function will be distinguishable for overload resolution purposes from the function generated for other actual arguments.

A template parameter used only in a default argument, a sizeof, or in the array bounds of the first array dimension are not considered to be used in the type. A type used in the return value of a function template is not considered to be used, but a type used as the return type of a function pointer argument is considered to be used.

```
template <class T> void f(int a[sizeof(T)]){}   // Not used
template <class T> T f(){}                      // Not used
template <class T> void f(T* (*fp)()){}         // Used
```

Status: Clarification of WP—not yet reviewed

Version added: 1
Version updated: 1

3.6 Question: Can unnamed types be used as template arguments?

```
struct {} a;
static union {} b;
template <class T> void f(T){}
void main()
{
        f(a);   // Error
        f(b);   // Error
}
```

Answer: No.

Status: Open

Version added: 1
Version updated: 1

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

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

Answer: Yes—but names that are types must be explicitly designated as types by whatever mechanism is invented to accomplish this (see 6.7).

Status: Open

Version added: 1
Version updated: 1

3.8 Question: Can a noninline function template be instantiated when referenced?

Answer: No. A specific definition of the function may be supplied in a separate file or even in a library supplied at link time. Furthermore, the template version of the function may not be able to be instantiated with a given set of actual template arguments.

This requires a more detailed description. In some environments it may be possible for a compiler to instantiate an instance of a function template and flag it in such a way that the linker will ignore the template definition in favor of a specific definition when the program is linked. As long as the compiler can suppress errors for instances that cannot be successfully generated, then such an environment would behave as if the required template instances were generated at link time.

Status: Clarification of WP—not yet reviewed (specialization issues are still under discussion).

Version added: 1
Version updated: 1

3.9 A proposal to allow `Derived<T>` to `Base<T>` conversions in function template calls.

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 this sort of conversion 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 `Derived<T>` to `Base<T>` conversion requires, 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.
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*` $\Rightarrow$ `B*`) that is less desirable than the one for the nontemplate `f` (`F*` $\Rightarrow$ `D*`). Likewise, the template `g` should be chosen because on the second argument the nontemplate `g` requires a standard conversion (`F*` $\Rightarrow$ `B*`) that is less desirable than the one for the template `g` (`F*` $\Rightarrow$ `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*` $\Rightarrow$ `B<int>*`) that is less desirable than the one for the nontemplate f (`F*` $\Rightarrow$ `D<int>*`). Likewise, the template `g` should be chosen because nontemplate `g` requires a standard conversion (`F*` $\Rightarrow$ `B<int>*`) that is less desirable than the one for the template `g` (`F*` $\Rightarrow$ `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);
}
```

Member Function Templates

4.1 Question: Are inline member functions that are not used by a given class template instance instantiated?

Answer: No—inline member functions are instantiated when they are first used.

Status: Open

Version added: 1
Version updated: 1

4.2 Question: Can a noninline member function or a static data member be instantiated when referenced?

Answer: No. A specific definition of the member function or static data member may be supplied in a separate file or even in a library supplied at link time. Furthermore, the template version of the member function or static data member may not be able to be instantiated with a given set of actual template arguments.

This requires a more detailed description. See the note regarding instantiation of noninline function templates (3.8) above.

Status: Clarification of WP—not yet reviewed

Version added: 1
Version updated: 1

4.3 Question: Must the template parameter names in a member function definition match the names used in the class definition?

```
template <class T1, class T2> struct A {
        void  f1();
        void  f2();
};

template <class T2, class T1> void A<T2, T1>::f1(){}   // OK
template <class T2, class T1> void A<T1, T2>::f2(){}   // error
```

Answer: No—different names may be used but the template parameters must be used in the template argument list of the function declarator in the same sequence in which they were declared in the template parameter list.

Status: Open

Version added: 1
Version updated: 1

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

4.5 Question: How are default arguments for parameters of member functions of class templates handled?

Answer: They are handled similarly to the nontemplate case. Additional default arguments can be supplied on subsequent declarations and definitions of the functions.

When a class template member function is defined outside of the class any additional default arguments are added to the class template and are also added to the generated instantiations that have been created so far.

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

A<float> af;
A<int>   ai;

void A<int>::f(int i, int j = 1){}

// The next line updates the default argument information for
// A<T>::f, A<float>::f, and A<int>::f.  An error will be
// issued because A<int>::f already has a default argument
// for argument j.
template <class T> void A<T>::f(int i = 1, int j = 0){}

void A<char>::f(int i, int j = 1){}  // Redeclaration of default?
```

```
void main()
{
        af.f(1);      // OK
}
```

Status: Open

Version added: 1
Version updated: 1

Class Template Specific Declarations and Definitions

5.1 Question: Can you create a specific definition of a class template for which only a declaration has been seen?

Answer: Yes.

Remarks: This may be affected by the outcome of the discussions of the rules for creating specific definitions of class templates. For example, there may be a requirement that a complete template definition be available so that the specific definition may be compared with the template to ensure that it conforms to certain consistency requirements.

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

Status: Open

Version added: 2
Version updated: 2

5.2 Question: Can you declare an incompletely defined object type that is a specific definition of a class template?

Answer: Yes. In this example `struct A<int>;` declares an incompletely defined object type called `A<int>`. It does not cause an instantiation to be generated from class template `A`.

```
template <class T> struct A {};
struct A<int>;
A<int> a;   // Error - A<int> is an incomplete type
```

Status: Open

Version added: 2
Version updated: 2

5.3 Question: Can the class template name be used as a synonym for the current specific definition inside the specific definition?

Answer: Yes—the following two examples are equivalent. See also 2.2.

Example 1:

```
struct B<int> {
        B<int>(){};  // Allowed - change from WP
        ~B<int>(){}; // Allowed - change from WP
        int B<int>::* pmi;
        B<int>*   bp;
        B<int> f1(B<int>& b);
        B<int> f2(B<int>& b) {B<int> b2 = b;return b2;}
        struct C {
                int B<int>::C::* pmi;
                B<int>::C*       bp;
                B<int>::C f2(B<int>::C& b)
                {
                        B<int>::C b2 = b;return b2;
                }
        };
        friend B<int> g(B<int>& b);
};
B<int> B<int>::f1(B<int>& b) {B<int> b2 = b; return b2;}
```

Example 2:

```
struct B<int> {
        B(){};
        ~B(){};
        int B::* pmi;
        B*   bp;
        B f1(B& b);
        B f2(B& b) {B b2 = b;return b2;}
        struct C {
                int B::C::* pmi;
                B::C*       bp;
                B::C f2(B::C& b)
                {
                        B::C b2 = b;return b2;
                }
        };
        friend B g(B& b);
};
B<int> B<int>::f1(B& b) {B b2 = b; return b2;}
```

Note that member functions defined in the class body and member functions defined outside of the class are handled equivalently.

Status: Open

Version added: 2
Version updated: 2

5.4 Question: Can a specific definition of a class template be a local class?

Answer: No. A class template declaration declares a set of global scope classes. Specific definitions of members of that set of classes must also be declared and defined at global scope.

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

int main()
{
        struct A<int> {};  // Error
}
```

Status: Open

Version added: 2
Version updated: 2

Other Issues

6.1 Question: Should classes used as template arguments have external linkage?

Answer: Yes – by inference from the prohibition against using the address of an object or function with internal linkage as a template argument (14.2), it would seem that "used as a template argument" should be added to the list of attributes that force a class to be externally linked (3.3).

Status: Open

Version added: 1
Version updated: 1

6.2 Question: When must errors in template definitions be issued and when must they not be issued?

Answer: If a template contains errors that make it impossible for any valid instance to be generated an error may be issued when the template definition is processed. An implementation is not required to issue errors at this point, however. Errors may be deferred until an instantiation is required.

When processing a template that can be instantiated with certain arguments but not with others, an error must not be issued unless the offending instantiation is required.

```
template <class T> struct A {
        int     a;
        char    a;  // Always an error -- may be issued when the
                    // template is scanned or at instantiation
};

A<int>  a;      // Previous error may be issued here

template <class T> struct B {
        T       a[10];
        void f() { T    a[10]; }
};
```

```
B<int&> b;    // Error instantiating class B -- array of reference
B<char&>* b2; // No error - does not cause complete
              //             instantiation of B<char&>

void main()
{
        b.f(); // Error instantiating B<int&>::f
               // array of reference
}
```

If the call of `b.f()` were eliminated, the error must not be issued. What if only the `b2` is declared? This does not require an instantiation of a completely defined object type and so an error should not be generated.

Remarks: There are those that would like the draft to specify that certain errors must be diagnosed in a template even in the absence of any instantiations of that template. We cannot currently require compilers to parse arbitrary template definitions in the absence of an instantiation because some templates cannot be parsed without additional information about which names are or are not types. This is one of the template issues that must be resolved—but without a resolution of this issue we cannot require compilers to perform a certain level of analysis of unreferenced templates.

Even when the type/nontype issue is solved providing a specification of which errors must be diagnosed while scanning unreferenced templates is an extremely difficult job. Doing so would require that we

- specify which errors are required to be issued in an unreferenced template and which errors are not;
- specify the conditions under which certain errors are or are not required to cause errors to be issued (certain errors can only be detected in contexts that do not involve template parameters or names from a base class that is specified using a template parameter).

This information is not yet specified for the language as a whole, so we are certainly not in a position to attempt to specify this for template definitions.

I would propose that the recommendation made above be adopted and that the issue not be revisited until we have some implementation experience with whatever mechanism is used to specify whether a name is a type or nontype.

Status: Open

Version added: 1
Version updated: 2

6.3 Question: What kinds of types may be used in a function template declaration while still being able to deduce the template argument types?

When a call to an instance of a function template is scanned the compiler is required to infer the types of the template arguments from the types of the function arguments.

In this example the compiler must infer from the call of function `f` that the type of `T1` is `char*` and the type of `T2` is `int`.

```
template <class T> struct A {};
template <class T1, class T2> void f(A<T1>, void (*f)(T2)) {}

void xyz(int){}

void main()
{
        A<char*> a;
        f(a, xyz);
}
```

Answer: Each function template parameter type may comprise any combination of the following elements:

```
T
cv-list T
T*
T&
T::member-type-name     // Must be explicitly designated as a type
T[integer-constant]
class-template-name<T>
type (*)(T)
T class-name::*
type T::*
```

Status: Tentatively approved by the extensions working group in Portland.

Version added: 1
Version updated: 2

6.4 Question: Can a static data member of a class template be declared with an incomplete array type?

In the following example when is the size of the static data member `i` known?

```
template <class T> struct A {
        enum { xxx = sizeof(T) };
        static int i[];  // Error
};

template <class T> int A<T>::i[xxx] = {1};
int A<double>::i[xxx] = {1};

int main()
{
        A<int> a;
        A<double> a;
        int i = sizeof A<int>::i;     // Error - size unknown
        int j = sizeof A<double>::i;  // OK
}
```

Answer: No—A static data member in a class template may not have an incomplete array size.

Rationale: In the example above the size of `A<int>::i` cannot be known because a specific definition may be provided in another file. The size of `A<double>::i` is known after its specific definition has been seen. Users are likely to be confused by the difference in the way these two examples would have to be handled. Rather than introduce a confusing rule the suggestion is to specify that a static data member in a class template may not have an incomplete array size.

The alternative is to specify that the size of a static data member whose array size is incompletely defined by its class definition is only known if a specific definition of the static data member has been seen in the compilation being processed.

This would make implementation more complex. When an implementation instantiates `A<int>::i`, it may do so as part of a process that includes the instantiation of other functions. If one of the other functions references the size of `A<int>::i` it should cause an error to be issued indicating that the size of `A<int>::i` is unknown even though at that point the size might actually be known. This error would need to be issued to ensure that a program is handled equivalently regardless of the sequence in which its templates are instantiated.

This is related to the issue of requiring a certain level of consistency between a class template and its specific definitions.

Status: Open

Version added: 2
Version updated: 2

6.5 Question: How should template arguments that contain "`>`" be parsed?

Answer: The first `>` in a template argument terminates the template argument list. To use a `>` in a template argument the argument must be in parentheses (or other nested construct). This is equivalent to the handling of comma operators in the context of a function argument list.

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

int main()
{
        A< 1 > 2 > a1;  // Error
        A< (1>2) > a2;  // OK
}
```

Nesting of template references is allowed as illustrated in this example:

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

int main()
{
        B< A<1> >  a1;  // OK
}
```

Status: Open

Version added: 2
Version updated: 2

6.6 Question: Can template versions of `operator new` and `operator delete` be declared?

Answer: Only the multiple argument version of `operator new` may be declared.

```
template <class T> void* operator new(T);          // Error
template <class T> void operator delete(T);        // Error
template <class T> void* operator new(size_t, T);  // OK
```

Rationale: Defining the single operand version of `new` or `delete` with a template is pointless because there is only one valid signature. Furthermore, unless the template definition were declared inline there is no way that the template version would ever be used—the library supplied version would be viewed as a specific definition that would be used in preference to the template defined version.

Status: Open

Version added: 2
Version updated: 2

6.7 Question: 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?

Answer: An unknown name is considered to be a nontype unless explicitly designated as a type (as has previously been proposed by Bjarne and others).

There are two possible means of explicit designation. The first is to use the keyword `class` and the second is to add a new keyword such as `typename`.

Using the `class` keyword to distinguish types from nontypes create an ambiguity between an explicit type designation and a forward declaration of a nested or local class as illustrated in the following example:

```
template <class T> struct A : public T {
        xyz     a;      // Error - xyz is not a type
        T::xyz  a;      // Error - xyz is not a type
        class B;        // Ambiguous
        class T::abc;   // OK - T::abc is a type
};
```

The ambiguity can be avoided by constraining the kinds of names that may be used in an explicit type designation. We could require that the name in an explicit type designation be a qualified name, such as `T::abc` in the example above. A consequence of this requirement would be that global names could not be explicitly designated using this mechanism.

```
template <class T> struct A : public T {
        class B;        // Forward decl. of nested class
        class T::abc;   // OK - T::abc is a type
        // No way to designate a global name as a type
};
```

Another potential problem with using the `class` keyword is the potential for confusion between the use of `class` to mean "type" and use of `class` as an elaborated type specifier.

The alternative is to use a new keyword such as `typename`. The example would then be rewritten as:

```
template <class T> struct A : public T {
        class    B;        // Forward decl. of nested class
        typename T::abc;   // OK - T::abc is a type
        typename ::X;      // OK - ::X is a type
};
```

The disadvantage of this approach is, of course, that it requires a new keyword which has the potential of affecting existing programs that use the keyword as an identifier.

Status: Open

Version added: 1
Version updated: 2

Nontype Parameters for Function Templates

In Portland the `Bitset` class was voted into the draft. The `Bitset` class requires that function templates be allowed to have nontype parameters.

This is a proposal of a minimal set of facilities required to provide the necessary support for nontype parameters. Nontype parameters of function templates differ from nontype parameters of class templates because they must be deduced from the actual arguments of the functions where class template parameters are explicitly specified. The minimal set of facilities is designed to avoid situations in which it is difficult or impossible to deduce the appropriate value for a nontype parameter.

Nontype parameters of a function template may only be used to specify the nontype arguments of a class template or as an array bound. As with type parameters, every nontype parameter must be used in at least one function parameter type declaration. For example,

```
template <int I> void f(A<I>);  // OK
template <int I> A<I> f();      // Error - I not used in a parameter type
template <int I> A<I> f(B<I>);  // OK
template <int I> void f(int array[10][I]); // OK
template <int I> void f(int array[I][10]); // Error - major array bound
                                           // not part of parameter type
```

Nontype parameters may not be used in expressions in the function declaration. The type of the function template parameter must match the type of the class template parameter.

```
template <int I> void f(A<I+1>);  // Error - expressions not allowed

template <char C> class A {};
template <int I> void f(A<I>);    // Error - conversion not allowed
```

Nontype parameters may not have default arguments. Because function template parameters are deduced from the function arguments there is no need to allow defaults on both the function parameters and the template parameters. The name of a nontype parameter may not be omitted because there would be no way of deducing the value of the omitted parameter.

Status: Open

Version added: 2

Version updated: 2

Other Issues—Without Resolutions

These are, in general, some of the larger and more intractable of the template issues. Most of these are discussed in Bjarne Stroustrup's paper "Major Template Issues" (93-0081/N0288).

1. Constraints on actual arguments.
2. Name binding—including operators
3. Instantiation of noninline template functions, member functions of template classes and static data members of template classes
4. Template overloading
5. Explicit qualification of template function calls
6. Specializations.