Major Template Issues
(Note for discussion. Revision 0)

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This note discusses major unresolved issues related to templates. Most issues have been discussed before on the -ext reflector, in notes, and at meetings. Most are marked by editorial boxes in the WP. I expect that further discussion will take place on the -ext reflector so that I can have a revised version of this note ready as a basis for extensions group work at Valley Forge. This note is (necessarily) incomplete. Unfortunately, time pressure has made it even more incomplete that it ought to be. Sorry. Few topics – if any – will be new to people who have followed the discussions of template issues closely, yet I expect that I have pulled enough strands together to provide something new to most.

1 Introduction

I have tried to order these issues in order of importance. This is necessarily subjective, but please note that the first three issues:

[1] typename qualification
[2] compilation model
[3] name injection

must be resolved for the language specification to be complete. I think all ought to be resolved at Valley Forge.

In each case, I have suggest a resolution. In each case, I expect a debate leading to either a firmer conclusion or a better alternative. In each case where the debate leads to a tentative conclusion that a change to the WP is needed, I'd like to see draft WP text ready before the Valley Forge meeting. Please email me comments ASAP so that I can have a revision of this note ready for then.

2 typename qualification

Section [temp.res] of the post-Waterloo WP says:

‘‘A name used in a template is assumed not to name a type unless it has been explicitly declared to refer to a type in the context enclosing the template declaration or in the template itself before its use. For example:

```
        // no B declared here

        class X;

        template<class T> class Y {
                class Z; // forward declaration of member class
                typedef T::A; // A is a type name

                void f() {
                        X* a1;    // declare pointer to X
                        T* a2;    // declare pointer to T
                        Y* a3;    // declare pointer to Y
                        Z* a4;    // declare pointer to Z
                        T::A* a5; // declare pointer to T’s A
                        B* a6;    // B is not a type name:
                                  // multiply B by a6
                }
        };
```

’’

The principle that a name names a type if and only if it has been explicitly declared to (and names a template only if it has been explicitly declared to) is important. Without it, syntax analysis of C++ would need to interact closely with semantic analysis – and would probably be impossible even in relatively simple cases.

However (quoting directly from an editorial box),

‘‘There is a potentially serious problem with the rule that a name is a non-type name unless it has been explicitly specified to be the name of a type. In some contexts, there is no way of indicating that a name is a type name before using it. For example:

```
        template <class Predicate> class unary_negate
                : unary_function<Predicate::argument_type, int>, // syntax error!
                  restrictor<Predicate>
        {
                // ...
        };
```

There are two alternative solutions: (1) Allow the compiler to assume that a qualified name is the name of a type in ‘‘such contexts,’’ thus making the example above well-formed as written. (2) Introduce a keyword to allow the programmer to state that a name is a type name as part of using that name. For example:

```
        template <class Predicate> class unary_negate
                : unary_function<typename Predicate::argument_type, int>,
                  restrictor<Predicate>
        {
                // ...
        };
```

Solution (2) would replace the *type-name-declaration* construct.’’

The `typename` qualifier would be optional in almost all cases but essential in a few key cases, just like the `class` qualifier is.

When processing Jon Spicer’s list of template issues (#6.7), we considered a less general variant of `typename`, but preferred (at my recommendation, sorry) the less radical solution of ‘‘recycling’’ `typedef` as in the first example in this section. Unfortunately, that solution appears to have been insufficient.

I prefer a new keyword to a rule introducing context sensitivity (solution (1) above) because

[1] it is explicit, and

[2] it is general (can be used wherever we need to name a type), and

[3] it makes to novel use of `typedef` redundant (I hope we will be able to eliminate the novel use of `typedef` before it takes root), and

[4] it would allow us to use `typename` instead of `class` in template declarations:

```
        template<typename T> class Vector { /* ... */ };
```

and

[5] it would break very little code.

The last reason is why I suggest `typename` rather than the nicer keyword `type`. Can you think of a better work to use as the keyword? Do we need an equivalent construct to say ‘‘this name denotes a template?’’ In theory we do, but I don’t see a practical need.

One reason I prefer the explicit qualification by `typename` approach to introducing context sensitivity is that I fear that a list of contexts where a name is a type name by default will be hard to remember for users – even if it turned out to be easy for us to produce.

With the `typename` keyword we could write the example from [temp.res] like this:

```
// no B declared here

class X;

template<class T> class Y {
        class Z; // forward declaration of member class

        void f() {
                X* a1;    // declare pointer to X
                T* a2;    // declare pointer to T
                Y* a3;    // declare pointer to Y
                Z* a4;    // declare pointer to Z
                typename T::A* a5; // declare pointer to T’s A
                B* a6;    // B is not a type name:
                          // multiply B by a6
        }
};
```

or even like this:

```
// no B declared here

class X;

template<typename T> class Y {
        class Z; // forward declaration of member class

        void f() {
                X* a1;    // declare pointer to X
                T* a2;    // declare pointer to T
                Y* a3;    // declare pointer to Y
                Z* a4;    // declare pointer to Z
                typename T::A* a5; // declare pointer to T’s A
                B* a6;    // B is not a type name:
                          // multiply B by a6
        }
};
```

However, allowing

```
// no B declared here

class X;

template<class T> class Y {
        class Z; // forward declaration of member class
        typename T::A;    // T’s A is a type

        void f() {
                X* a1;    // declare pointer to X
                T* a2;    // declare pointer to T
                Y* a3;    // declare pointer to Y
                Z* a4;    // declare pointer to Z
                A* a5;    // declare pointer to T’s A
                B* a6;    // B is not a type name:
                          // multiply B by a6
        }
};
```

would require a separate extension. Note that allowing only a construct like

```
typename T::A;    // T’s A is a type
```

would only be equivalent to

```
typedef T::A;    // T’s A is a type
```

and not solve the general problem. However, I do like that last variant/extension because it allows the specification of `T::A` as a typename to be done separately in the same way as I prefer

```
class X;
X* p;
```

to

```
class X* p;
```

in contexts where the two are equivalent. It is easy to miss the `class` qualifier in the midst of another declaration.

3 Template Compilation Model

We need a model of compilation of templates that tells people where to put their template declarations and definitions. Three main suggestions have been made, each with a bewildering number of variants:

[1] Implementation dependent: The ARM didn’t specify a compilation model. It simply stated that there had to be a – possibly implementation-defined – mapping between templates used and their definitions. This was – not too surprisingly – found to be a source of porting problems as different suppliers addressed issues of convenience, efficiency, etc., in different ways. It gave us lots of experience to work from, though.

[2] Repository: Let the user present the necessary template definitions to the compilation system and let the compilation system keep track of the correspondence between the templates used and the template definitions presented to it. The maintenance of the repository is extra-linguistic. In particular, the issues of time order and different versions are not addressed by the language. This is described in §3.1 below.

[3] Require the user to specify where definitions are to be found: Require that the user present the compiler with a ‘‘template directive’’ indicating where the template definition is found for each template used. This is described in §3.2 below.

Approach 3 was preferred by a small majority at the Waterloo, approach 2 by a small minority, and a large majority preferred to have more information before deciding.

An ideal approach would, among other things

[1] require the user to say as little as possible about where definitions are supposed to be found;

[2] allow the user to be specific about where definitions are supposed to be found;

[3] give reasonably efficient compilation and linking by default;
[4] be amenable to optimization of the compile and link process;
[5] be portable over all implementations;
[6] be easily comprehensible;
[7] allow good (that is, early) error detection and reporting;
[8] allow optional improve code generation (that is, inlining) based on knowledge of definitions;
[9] allow users to find definitions easily;
[10] not require yet-to-be invented techniques for its implementation.

3.1 A Repository-based Approach

Here is the repository approach presented in ‘‘Stroustrup: *The Design and Evolution of C++* Addison-Wesley 1994’’ and discussed in ANSI X3J16/94-0026, ISO WG21/N0413:

‘‘The concept of a central point where information related to templates and other issues that affect multiple compilation units must be recognized. I call that point ‘‘the repository’’ because its role is to keep information that the compiler needs between compilations of the separate parts of a program.

Think of the repository as a persistent symbol table with one entry per template that the compiler uses to keep track of declarations, definitions, specializations, uses, etc. Given that concept, I can outline a model of instantiation that supports all language facilities, accommodates the current uses of `.h` and `.c` files, doesn't require the user to know about the repository, and provides the alternatives for error checking, optimization, and compiler/linker efficiencies that implementers have asked for. Note that this is a model for an instantiation system, rather than a language rule or a specific implementation. Several alternative implementations are possible, but I suspect a user could ignore the details (most of the time) and think of the system this way.

Let me outline what might happen in a number of cases from the point of view of a compiler. As usual, `.c` are fed to the compiler and these `.c` files contain `#include` directives for `.h` files. The compiler knows only about code that has been presented to it. That is, it never looks around in the file system to try to find a template definition that it hasn't already been presented with. However, the compiler uses the repository to ‘‘remember’’ which templates it has seen and where they came from. This scheme can easily be extended to include the usual notions of archives. This scheme can easily be extended to include the usual notions of archives. Here is a brief description of what a compiler does at critical points:

– A template declaration is seen: The template can now be used. Enter the template into the repository.
– A function template definition is seen in a `.c` file: The template is processed as far as necessary to enter it into the repository. If it is already entered, we give a ‘‘double definition’’ error unless it is a new version of same template.
– A function template definition is seen in a .h file: The template is processed as far as necessary to enter it into the repository. If it is already entered, we check that the already entered template did in fact originate in the same header. If not, we give a ‘‘double definition’’ error. We check that the one-definition rule hasn't been violated by checking that this definition is in fact identical to the previous one. If not, we give a ‘‘double definition’’ error unless it is a new version of same template.
– A function template specialization declaration is seen: If necessary, give a ‘‘used before specialized’’ error. The specialization can now be used. Enter the declaration into the repository.
– A function template specialization definition is seen: If necessary, give a ‘‘used before specialized’’ error. The specialization can now be used. Enter the definition into the repository.
– A use is encountered: Enter the fact that the template has been used with this set of template arguments into the repository. Look into the repository to see if a general template or a specialization has been defined. If so, error checking and/or optimizations may be performed. If the template has not previously been used for this set of template arguments, code may be generated. Alternatively, code generation can be postponed until link time.
– An explicit instantiation request is encountered: Check if the template has been defined. If not, give a ‘‘template not defined’’ error. Check if a specialization has been defined. If so, give an ‘‘instantiated and specialized’’ error. Check if the template has already been instantiated for this

set of template arguments. If so, a ‘‘double instantiation’’ error may be given, or the instantiation request may be ignored. If not, code may be generated. Alternatively, code generation can be postponed until link time. In either case, code is generated for every template class member function presented to the compiler.

– The program is linked: Generate code for every template use for which code hasn’t already been generated. Repeat this process until all instantiations have been done. Give a ‘‘used but not defined’’ error for any missing template functions.

Code generation for a template and a set of template arguments involves the lookup at the point of instantiation mentioned in §???. Naturally, checking against illegal uses, unacceptable overloadings, etc., must also be performed.

An implementation can be more or less thorough in checking for violations of the one-definition rule and the rule against multiple instantiations. These are non-required diagnostics so the exact behavior of the implementation is a ‘‘quality of implementation’’ issue.’’

The primary benefit of this scheme is that the language definition need not concern itself with the organization of source text. The major snag is that *someone* has to worry about issues such as

[1] Is the definition being presented a new version or a double definition.

[2] When two projects share a repository – as they will *insist* on – how do a user tell if an object file has become invalid.

In addition, heuristics must be developed to avoid repository lookups from becoming a major performance bottleneck. In principle, these problems are solved. For example, the Ada world lives (happily?) with a scheme of this kind, but feedback from C++ users of repository schemes are not encouraging.

3.2 Template Directives

Here is the template directive variant proposed at the Waterloo meeting:

‘‘A template that has been used in a way that requires a specialization of its definition may be explicitly specialized (_temp.spec_) or explicitly instantiated (_temp.explicit_) within the current translation unit, otherwise the specialization will be implicitly generated if the definition has either been previously made available through a template-directive or has been previously defined in the current translation unit, else the template shall be explicitly instantiated within the program.

The syntax for a template directive is:

template-directive:
    `template` *string-literal* `;`

A template-directive nominates a translation unit containing definitions of templates that have been declared in the current translation unit. There shall be an implementation- defined mapping between the string-literal and an external source file name that is used to produce the nominated translation unit. The nominated translation unit is produced using the same environment as for the current translation unit. Names declared in the current translation unit are not visible within the nominated translation unit. Names declared within the nominated translation unit do not become visible within the current translation unit. After a template-directive has been processed, the template definitions within the nominated translation unit are available for use in instantiation of the declared templates if required. Definitions of objects or functions with external linkage within the nominated translation unit shall be well-formed but shall otherwise be treated as declarations only (i.e., no storage will be allocated and no code will be generated). A template-directive shall not appear in class scope or local scope. The treatment of definitions of objects or functions with internal linkage will be revisited when linkage issues are resolved.

In order to explicitly instantiate the specialization, the definition shall either be made available through a template-directive or shall have been previously defined in the current translation unit. For explicit instantiation of a class, the definition of all members shall be made available.’’

The main benefit of this proposal is that the definition of any template can be found by examining the program text. Its origins lies with the `#include` scheme used to find header files (See Spicer issue #6.20).

Making template definitions available through `#include` is from a language point of view equivalent to placing the template definitions in the source. That is too specific for many uses because we would like the template definitions to be as independent of a specific point of use as possible. It is also too inefficient for important applications and development environments. However, the idea of being able to point the compiler at the template source and then have it use that information if and only if it has needs for it is attractive.

Various degrees of precompilation of definitions made available through template directives is possible, and various degrees of optimization is possible based on the information (optimally) found as indicated by a template directive.

Like meaning the string specified in an `#include` directive, the string specified in a template directive is implementation specific. I don't expect that to be more of a problem than the equivalent problem with `#include`.

It has been suggested that a template derictive should be specific about which templates are defined where. For example:

```
template set "set.c"; // template set is defined in set.c
```

instead of plain

```
template "set.c"; // some template is defined in set.c
```

Heuristics will allow a compilation system to match the performance of the more specific version, and I suspect that the more specific version would become a nuisance by causing a proliferation of template directives and by constraining the way template definitions are stored. Also, once a more specific template directive was introduced the temptation to make it more flexible would be irresistible. For example:

```
template set "set1.c" "set2.c"   // template set is defined
                                 // in se1.c and/or set2.c.
template list map "containers.c" // list and map are defined
                                 //  in containers.c.
```

3.3 Context Merging

When a template specialization is generated the context for the resulting specialization must somehow be synthesized from the context of the template definition and the context of the use of the specialization.

<<I plan to expand this section>>

3.4 Recommendation

Accept some variant of the template directive proposal. I could live with the Waterloo proposal (§3.2).

4 Name Injection

Consider:

```
class complex {
        double re, im;
        friend complex operator+(complex,complex);
        // ...
};

void f(complex x, complex y)
{
        complex z = x+y;
}
```

The reason this works is that the name `operator+` is somehow ‘‘injected’’ into the scope enclosing the declaration of class `complex` so that `f()` can use it. We can debate the proper terminology for describing this phenomenon, but this is the way C++ has always worked, much code depends on it, and it is not easy to rewrite such code in a way that doesn't depend on some form of injection.

Now consider:

```
template<class Scalar> class complex {
        Scalar re, im;
        friend complex operator+(complex,complex);
        // ...
};

typedef complex<double> complex;

void f(complex x, complex y)
{
        complex z = x+y;
}
```

By analogy to the original example, most people – maybe naively – expects this to work, it did work the way templates were originally defined and accepted by X3J16/WG21, and significant code depends on it (Barton&Nachman???).

On the other hand, there is no doubt that name injection is odd (why should you have to look inside a declaration to figure out what is being declared?), and that name injection from template cause serious complications for the language definition and for implementors. Yet, it appears fundamentally odd injection for ordinary classes and not for template classes. If injection is allowed we must specify exactly how that is done. The extensions group and the committee as a whole have repeatedly changed their position on aspects of this issue (Spicer #2.23).

4.1 Problems with Injection

Consider this ‘‘classic’’ example of problems with name injection from templates, where the comments reflects the original resolution of the friend functions:

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

void g(void* fp)
{
        X* x;       // error: X undefined
        fp = f;     // error: f undefined
        f2(x);      // error: f2 undefined

        A<int> a;   // instantiate A<int>
                    // declare f(A<int>), f2(A<int>), and X somewhere

        X* x2;      // ok: X now defined
        fp = f;     // ok: f now defined

        A<char> b;  // instantiate A<char>
                    // declare f(A<char>), f2(A<char>), and X somewhere

        fp = f;     // error: ambiguous, f(A<int>) or f(A<int>)?
}
```

This resolution has the undesirable property that the interpretation of the statement `fp=f` changes as we progress down the function body – in a way that would appear completely mysterious in a real program.

We need to steer a course between these alternative absurdities.

4.2 Delayed Injection

John Spicer (among others) proposed a resolution that appears to both serve the sensible demands of users (which, because of existing code cannot be ignored anyway), and the equally sensible demands of implementors: Names from a template are injected *after* the global declaration in which a specialization is used. This would give this resolution:

```
void g2(void* fp)
{
        X* x;       // error: X undefined
        fp = f;     // error: f undefined
        f2(x);      // error: f2 undefined

        A<int> a;   // instantiate A<int>
                    // declare f(A<int>), f2(A<int>), and X after g2()

        X* x2;      // error: X undefined
        fp = f;     // error: f undefined

        A<char> b;  // instantiate A<char>
                    // declare f(A<char>), f2(A<char>), and X after g2()

        fp = f;     // error: f undefined
}

X* x3; // fine
void* fp2 = f; // error: ambiguous, f(A<int>) or f(A<int>)?
```

Leaving the ‘‘artificial’’ test case behind consider again a `complex` template. The key example

```
void f(complex x, complex y)
{
        complex z = x+y;
}
```

but introducing a specialization of `complex` for the very first time in a translation unit in the middle of a function would not:

```
void ff()
{
        complex<float> x = 1;
        complex<float> y = 2;
        complex<float> z = x+y;  // error:
                                 // operator+(complex<float>,complex<float>)
                                 // not declared
}
```

I think I could live with that. Does anyone have the means to check out a considerable amount of code to see whether that kind of example occur frequently? The nastiest variant would be the one where `x+y` was successfully resolved to another `+`, say `operator+(complex<double>,complex<double>),` by using conversions.

4.3 Delayed injection with re-scanning

The point of instantiation for a template specialization is ‘‘just before the nearest enclosing namespace declaration.’’ Typically, that is just before the function that first uses a specialization. Why don’t we just inject the names there – and analyze the program exactly as if the user had explicitly written those declarations. For example:

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

        // inject ‘void f(A<int>);’
        // inject ‘struct X;’
        // inject ‘void f2(X*);’
        // inject ‘void f(A<char>);’
```

```
void g2(void* fp)
{
        X* x;       // ok
        fp = f;     // error: ambiguous
        f2(x);      // of

        A<int> a;   // instantiate A<int>

        X* x2;      // ok
        fp = f;     // error: ambiguous

        A<char> b;  // instantiate A<char>

        fp = f;     // error: ambiguous
}
```

The snag is that this requires that we first scan a function body for instantiations and then do syntax and semantic analysis for it after every instantiation is done. Since injected names could determine whether a function was syntactically legal, this would be a nightmare. For example, the body of `g2()` contains a syntax error until the instantiation of `A<int>` has triggered the injection of `X`. Also, recursive instantiations could change the meaning of a program (syntactically and semantically) repeatedly. This is why we initially rejected this simple-minded model, and why I think it should stay rejected.

4.4 Immediate Injection

After I declare an ordinary class, I don't have to wait for some sort of instantiation to make injected declarations available. For example:

```
class X {
        friend class Y;
        // ...
};

Y* p;
```

works even when no use has been made of `X`.

Could templates be handled similarly? Consider:

```
template<class T> class X {
        friend class Y;
        friend class Z<T>;
        friend int f(T);
        // ...
};

Y* p1;          // error?
Z* p2;          // error
Z<int>* p3;     // error?
int i1 = f(2);  // error?

X<int> x;

Y* p11;         // ok
Z* p22;         // error
Z<int>* p33;    // ok
Z<char>* p44;   // error?
int i1 = f(2);  // ok
```

Consider what we logically *could* inject at the point of the template declaration and then consider what it would be best to do.

We clearly could inject names that are independent of the template parameter exactly as we would have done had we been declaring a non-template. For example:

```
template<class T> class X {
        friend class Y;
        // ...
};
        // possible: inject ‘class y;’

Y* p1;  // ok (possible)
```

We could inject template declarations for names that are dependent on a template argument. For example:

```
template<class T> class X {
        // ...
        friend class Z<T>;
        friend int f(T);
        // ...
};
        // possible: inject ‘template<class T> class Z;’
        // possible: inject ‘template<class T> int f(T);’

Z* p2;          // ok (possible)
Z<int>* p3;     // ok (possible)
int i1 = f(2);  // ok (possible)
```

This have a pleasing crude simplicity to it, and mirrors the way ordinary declarations ‘inject’ names. I have a nasty feeling, though, that I might have missed some important point. Please check my logic and think about the implications for users.

I think that an implementor ought to be able to diagnose this error:

```
int f;

template<class T> class X {
        friend int f(T);     // error: X can’t be instantiated
        // ...
};
```

The reason `X<T>` cannot be instantiated for any `T` is that every `f()` will clash with `::f`. This example must be considered whichever rule we adopt.

4.5 Recommendation

Sigh. I’m not sure. I think some form of injection in necessary. Without it we make some fairly standard ways of writing C++ code very difficult (see Barton and Nackman) and we break non-trivial amounts of real code in a way that is hard to fix. I can live with late injection (without re-scanning). However, I find immediate injection more attractive, and would encourage people to look for snags. If no serious snags are found, I’ll recommend immediate injection.

5 Template Overloading and Specialization

Consider a template a simple vector:

```
template <class T> class vector { /* ... */ };
```

We can specialize it for a specific type, say `bool`, like this:

```
class vector<bool> { /* ... */ };

vector<int> vi(10);  // the general template
vector<bool> vb(10); // vector<bool> user specialization
```

However, we cannot currently specialize it for a well-defined set of type, say all pointers:

```
template <class T> class vector<T*> { /* ... */ }; // can’t say this
```

```
vector<int> vi(10);   // use the general template vector<T>
vector<int*> vip(10); // use vector<T*> user specialization
```

Even though that seems a pretty simple and something many have wanted to do.

Now consider a `vector` like the one in STL:

```
template <class T, template <class U> class Allocator = allocator>
        class vector { /* ... */ };
```

If we need to have this container specialized for some type, such as bool, we can specialize on all template parameters:

```
class vector<bool,allocator> { /* ... */ };
```

However, we can't do what we really want to do; that is, specialize on the first argument and leave the second one free for the user to specify:

```
template <template <class U> class Allocator = allocator>
        class vector<bool, Allocator> { /* ... */ };

vector<bool> vb1(10);          // vector<bool,allocator> user specialization
vector<bool,myalloc> vb2(10);  // vector<bool,myalloc> user specialization
```

Similarly, we might want a partial specialization for vectors of pointers:

```
template <class T, template <class U> class Allocator = allocator>
class vector<T*, Allocator> { /* ... */ };

vector<int*> vi;  // use vector<int*,allocator> user specialization
```

5.1 Specialization as Overloading

Specialization is a rather crude form of overloading. Basically, the compiler can tell a completely specific example from the general case, but has no notion of a better match. The examples above suggests a way to allow a limited form of overloading by allowing specific specialization patterns. Note that this line of thought was not feasible until the rule that an explicit specialization must be declared before it is used was adopted. Before that, we had to consider overloading and specialization as alternative and interfering mechanisms.

Partial specialization could be made arbitrarily smart/general. That would, however, require the compiler to be arbitrarily smart or the introduction of rules for choosing between ‘‘similar’’ patterns. The other extreme – apart from the current acceptance of individual specializations only – would be to accept only a few simple patterns, such as `T*`. A better approach would be to accept exactly the set of patterns that define the set of acceptable template parameter deductions for functions from section [temp.deduct] of the WP. This would have the effect of allowing exactly the same set of overloadings for template functions and template classes. For example:

```
template<class T> class List { /* ... */ };
template<class T> class List<void*> { /* ... */ };
template<class T> class List<T*> { /* ... */ };
template<class T> class List< Vector<T> > { /* ... */ };
```

would provide distinct implementations of lists of `void*`, `T*`, and `vector<T>` and leave all other lists implemented by the general case.

Consider:

```
template<class T> class vector<T*> { /* ... */ };

vector<int*> vi(10);
```

What does `T` refer to for `vi`? `int` or `int*`? The answer is (clearly?) that in this case `T` stands for `int*` exactly as if there had been no specialization. The specialization pattern, in this example `T*` does not introduce a new variable or in any other way affect the meaning of the defined template; it simply controls when that template is used in instantiations.

Interestingly,

```
template<class T> class List { /* ... */ };
```

and

```
template<class T> class List<T> { /* ... */ };
```

become synonyms.

Should the general form of a template be required at all? The current rule is that you cannot specialize a template that hasn't been declared (because in that case there isn't a template to specialize). However, having a restricted form of some templates only appears to make perfect sense for some templates. For example:

```
template<class T> class X<T*> { /* ... */ };
```

would be a template that could be used for pointers only. However, I would like to see some much better examples before seriously considering relaxing the requirement that the general template be declared first of all. Without it, spelling mistakes in specialization declarations cannot be caught.

5.1.1 Specialization and Constraints

There have been persistent and repeated requests for a way of expressing constraints on template arguments. A discussion of the pros and cons of various forms of constraints on template arguments can be found in §15.4 of Stroustrup: *The Design and Evolution of C++* Addison-Wesley 1994. It has also repeatedly been noted that overloading is an alternative to constraints in many cases. That is, instead of saying ‘‘the parameter must have property X’’ one says ‘‘if the parameter has property X use this version otherwise use another.’’ The partial specialization technique would extend that technique to deal with class templates.

In *The Design and Evolution of C++*, I observe that no constraint mechanism (that I know of) can express every desirably constraint directly and conveniently. The partial specialization mechanism is no exception. For example, it is easy to express ‘‘T must be a pointer,’’ but one cannot directly express ‘‘T must not be a pointer.’’ In this case, the general template can be used to indirectly express the negative:

```
template<class T> class X { /* ... */ };     // non-pointer arguments only
template<class T> class X<T*> { /* ... */ }; // pointer arguments
```

The specialization for pointers makes the comment on the general case true.

Partial specialization according to the rules outlined above cannot express one of the most commonly requested constraints ‘‘the template argument must be a pointer to a class derived from `Base`.’’ One could extend the expressiveness of specialization patterns to cover that case:

```
template<class T> class Z { /* ... */ };

template<class T> class Z<T*:B*> { // specialization for T*s where
                                   // T is derived from B
        // ...
}
```

I am not keen on doing that, though. I have severe doubts about the kind of system architecture that results from the use of such constraints, I like the fact that the rule for specialization patterns is exactly the existing rule for template parameter deduction, and I like not having to invent a syntax for expressing derivation constraints.

5.2 Template Functions

We can overload a single template by as many functions as we like, and we can specialize a template function as often as we like, but we cannot define two function templates with the same name. This is a result of the current rules rather than of anything fundamental.

What would be the equivalent to partial specialization for template functions? Here is a first approximation of an answer: Overloading of a template function

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

with another with the argument of a form that allows the deduction of the template parameter according to the rules from [temp.deduct]. For example:

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

In a call, a pointer argument causes the invocation of the pointer version; non-pointer arguments cause the invocation of the general version. For example:

```
void g(int i, int* pi)
{
        f(i);  // call f<int>(int)
        f(pi); // call f<int*>(int*), the f declared f(T*)
}
```

This shows that the second function, `f(T*)`, is really a specialization of the first, `f(T)` because both could be called for `pi` and in both cases the template parameter would be `int*`; that is, in both cases the function called would be `f<int*>`. What would be done to get the example to work as described would be to prefer `f(T*)` over `f(T)` because `T*` is more specific, in exactly the way a specific user-defined specialization is chosen. For example:

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

void gg(int i, int* pi)
{
        ff(i);   // call f<int>(int)
        ff(pi);  // call f<int*>(int*), the f  declared f(int*)
}
```

5.2.1 Specialization and Linkage

Following this line of thought, it would be logical to require the specialization syntax to be used:

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

That is, however, rather ugly and apparently redundant. Is it logically necessary? If both functions are within a single compilation unit, it is clearly not, but how about:

```
// file1.c:

        template<class T> void f(T);

        void g(int* pi) { f(pi); }

// file1.c:

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

        void h(int* pi) { f(pi); }
```

How would a compilation system know that the two `f()s` were different functions and correctly find/generate their definitions? After all, each compilation knows only one template called `f` and each calls `f(int*)` with the template parameter `int`; that is, `f<int>f(int*)`. I conjecture that few existing implementation distinguish between the two cases (as, of course, they are not required to do), and that it would be non-trivial to change implementations to distinguish them because of link compatibility concerns. Thus, I conclude that even though specialization and overloading are equivalent (and trivial) within a single compilation unit, there is a serious problem to consider if one wants to allow either and both in a multiple compilation unit system (as one must). This will come as no surprise to people acquainted with the history of overloading and linking in C++; the two topics are intimately intertwined.

The way this problem is addressed for specializations is first to require the template to be specialize present before specialization can take place (this is violated in `file2.c` above), and then to require each specialization explicitly declared to be a specialization. This would give:

```
// file1.c:

        template<class T> void f(T);

        void g(int* pi) { f(pi); }

// file1.c:

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

        void h(int* pi) { f(pi); }
```

The alternative, used for function overloading, is for each template function call to carry enough information around at link time to distinguish it from a call of any other template function. That solution does not appear feasible to me.

The use of the abbreviated form `f<>` to indicate specialization where the template parameter can be deduced avoids the annoying redundancy and makes the notational overhead tolerable (to me at least).

Note that a similar notational convenience is neither needed nor possible for class templates because for a class template there are no function parameters from which the template parameters could be deduced, and thus no annoying redundancy to eliminate.

5.2.2 Partial Specialization for Functions

I conclude that we can allow partial specialization for template functions, but not simple overloading by other template functions. A partial specialization must have its template parameter deducible according to the usual rules. A partial specialization must be declared after its general template has been declared and must be declared before used – exactly as the specializations we have now.

The first approximation for resolution of calls is also exactly the same as the current rule: If a specialization matches, it is preferred to the general template. The rules for conversions (or rather the lack thereof) are the ones for arguments used to deduce a template parameter.

The remaining problem is what to do if two or more specializations match. I propose that a double match is an ambiguity except that `const` may be used as a tiebreaker. For example:

```
template<class T> void f(T);          // general template
template<class T> void f<>(T*);       // one specialization
template<class T> void f<>(const T*); // another specialization

void g(const char* pcc, char* pc, char c)
{
        f(pcc); // call f<char>(const char*)
        f(pc);  // call f<char>(char*)
        f(c);   // call f<char>(char)
}
```

I see no examples that tempt me to complicate the rules further.

5.2.3 Examples of Partial Specialization of Functions

Consider a copy function similar to the one in STL:

```
template <class T> T* copy(T* first, T* last, T* result)
{
        while (first!=last) *result++ = *first++;
        return result;
}
```

This is a rather inefficient way of moving simple objects, such as pointers, on many systems.

To avoid this overhead, we could define a specialization for the (important) case where `first`, `last`, and `result` points to pointers:

```
template <class T> inline T** copy<>(T** first, T** last, T** result)
{
        return (T**)memmove(result, first, (last - first)*sizeof(T*))
                + (last - first);
}
```

This will guarantee that no copy functions will be generated for all pointer moves; moreover, the performance of copy will be much better.

The same kind of technique can be used for sort, etc.

The technique also solves the nagging ‘‘swap problem.’’ A general `swap()` must be defined like this:

```
template <class T> void swap(T& a, T& b)
{
        T temp = a;
        a = b;
        b = temp;
}
```

If we have two `vectors` `a` and `b`, `swap(a,b)` does three vector assignments, and thereby copies `3*n` elements where `n` is the number of element of a vector. However, pointers or references to vectors can be swapped much more efficiently:

```
template <class T> inline void swap<>(vector<T>& a, vector<T>& b) {
        a.swap(b);
}
```

which is a constant time (and very fast) operation. That is important when you sort a vector of vectors.

5.3 Recommendation

I’m not sure. This mechanism seems right and addresses several important and well-known problems. In particular, the definition and implementation of STL would benefit immediately and significantly. It also seems to fit in with the rest of the language so that key syntax, linkage, type deduction, and overload resolution simply follows existing rules. On the other hand, it is not absolutely essential for the completion of C++. If *any* new features are considered this one ought to be.

6 Nested templates

Are there residual problems with nested templates? I would like to be sure that the answer is ‘no.’ Here, I present an example of a nested conversion function that – to my mind – indicates that a restriction might be needed (though I’m not sure and don’t propose one). I also present a nested template class example to give anyone who think there might be problems lurking here a chance to air their worries; in this area, I do not suspect problems.

6.1 Nested Conversions

One of the reasons to allow member templates was that several problems could be solved using conversions defined as member templates (see ‘‘Stroustrup: *Suggestions for Completing Templates* ANSI X3J16/94-0026, ISO WG21/N0413.’’ or ‘‘Stroustrup: *The Design and Evolution of C++* Addison-Wesley 1994.’’ However, we soon discovered that the mechanism seems to be more powerful than apparently necessary and that truly amazing examples could be constructed. For example:

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

template<class T> class Y {
public:
        template<class U> operator D<U>(); // convert any Y<T> to any D<U>
};

void f(B<int*>);

void g(Y<alpha> y)
{
        f(y);   // f( (B<int*>) y. Y<alpha>::operator D<int*>() )
}
```

This is a mild example. It can be handled by observing that one can get a `B<int*>` from a `D<int*>` and then noting that a `D<int*>` can be obtained from any `Y<T>` by the user-defined conversion. Should a compiler be that smart? Yes, I think so. In this case all that is required is a simple working backwards from the desired parameter type towards the argument type.

Consider this variant:

```
template<class T> class D;

template<class T> class Y {
public:
        template<class U> operator D<U>(); // convert any Y<T> to any D<U>
};

template<class T> class D : Y<T> { };

D<int*> x;

Y<double> y = x; //
```

To get a `Y<double>` we need a `D<double>`. A `D<double>` we can get from a `Y<anything>`, in particular, we can get it from a `Y<int*>`. We can get a `Y<int*>` from a `D<int*>` by simple derivation. Thus we can get from the `D<int*> x` to the `Y<double> Y`. I don't like that one little bit, and would like to outlaw it.

What I don't like is finding the user defined conversion based on the fact that there is a built-in conversion that can be used for the result of one particular user-defined conversion. In the first example above, the algorithm proceeded smoothly by through steps each looking for a single conversion needed to reach a known target. The second example seems to go beyond that and requiring the algorithm to look two conversions ahead. If this could be made precise, I think we have a basis for a rule that excludes absurdities without interfering with desirable conversions.

<<I plan to add to this section>>

6.2 Nested Class templates

Consider:

```
template<class A> class X {
        template<class B> class Y {
                // ...
        };
};
```

Does this imply problems about naming or instantiation? I don't see any. To name an inner class `Y<b>` you must name some outer class `X<a>` so that the context for the instantiation for `Y<b>` is determined exactly as if it had been defined within an ordinary class. For example:

```
class Xa {
        template<class B> class Y {
                // ...
        };
};
```

Thus, all questions of scope and instantiation reduces to previously solved problems.

6.3 Recommendation

Look carefully at examples of conversions defined as member templates. If a rule that outlaws the most outrageous examples (apply to Erwin Unruh if you want some even weirder ones :-) can be found we would probably be wise to adopt it. If not, do nothing.

7 Namespace Templates

Often, a class template carries no data, only static functions. For example:

```
template<class T> class CMP {
public:
        static int eq(T a, T b) { return a==b; }
        static int lt(T a, T b) { return a<b; }
};
```

The class itself and our use of it indicate that we don't need any of the aspects of the class concept relating to creating and using objects. It therefore ought to be a namespace:

```
template<class T> namespace CMP {
        int eq(T a, T b) { return a==b; }
        int lt(T a, T b) { return a<b; }
};
```

Allowing namespaces to be templates is a simple and logical extension.

When discussed in the extensions working group the only real worry about the soundness of namespace templates was whether the openness of namespaces would cause problems. I would address that concern by closing a namespace defined as a template:

```
template<class T> namespace X { /* ... */ };
template<class T> namespace X { /* more */ }; // error: can't add to template
```

7.1 Recommendation

This feature is nice, but not necessary. I think we should forget about it for now. If people really need/want it they will scream for it during the review period. I think they ought to have more important things to worry about.

8 Namespace Template Arguments

A namespace cannot be a template argument. The reason is purely historical – namespaces didn't exist (except as an idea to be explored) when I wrote the template text. The implication is that if I want to refer to a namespace from within a template I must either hard-wire the name of the namespace in or I must provide a global namespace alias. For example:

```
template<class T> void f(T)
{
        // ...
        Lib::g(); // calls the library
                  // ''Lib'' hard-wired into code
        // ...
}
```

or

```
namespace fLib = Lib; // f() used fLib

template<class T> void f(T)
{
        // ...
        fLib::g(); // calls the library
        // ...
}
```

The obvious alternative is to allow the last example to be expressed directly:

```
template<class T, namespace fLib> void f(T)
{
        // ...
        fLib::g(); // calls the library
        // ...
}
```

When discussed in the extensions working group the only real worry about the soundness of namespaces as template parameters was wether the openness of namespaces would cause problems. For example:

```
namespace A { void f(); }

template<namespace N> class X {
        // ...
}

        // ...

namespace A { void g(); } // add to A

        // ...
```

The definition of `X<A>` would depend on whether it was instantiated before or after the extension of `A`. I see this as a variant of the problem we have when people define different variant of an overloaded function in different places. Like a namespace, a set of overloaded functions is open. Thus, I don't see the openness of namespaces as a reason not to allow namespaces as template parameters.

8.1 Recommendation

This feature is nice, but not necessary. I think we should forget about it for now. If people really need/want it they will scream for it during the review period. I think they ought to have more important things to worry about.

9 Typedef Templates

Typedefs for templates are useful. For example:

```
template<class T1, class T2> class X { /* ... */ };

typedef X<int,char> xic;

xic x0;
```

However, a `typedef` refers to a specific type and every parameter for the template must therefore be specified. It is not possible to use a `typedef` to give a name to the template that arise from specifying one argument and leaving the other free for the user to specify later. This could be allowed. For example:

```
template<class T> typedef X<T,char> xc;

xc<int> x1;  // same type as xic above.
xc<char> x2;
```

One nit is that this would be using `typedef` to name something that is not a type.

9.1 Recommendation

This feature is nice, but not necessary and not 100% clean. Most of its intended functionality is provided by the more general mechanism of partial specialization, and by simple derivation. I think we should forget about it for now. If people really need/want it they will scream for it during the review period. I think they ought to have more important things to worry about.

10 Syntax of Template Types

Writing something like

```
        vector<list<char>> vlc;
```

must be one of the most common and most annoying errors done by both novices and experts. The detection and repair is trivial:

```
        vector< list<char> > vlc;
```

However, I still find the first – buggy – version the most visually pleasing. After years, I still haven't become really reconciled with the second.

We have discussed the problem with that trailing >> often enough, I don't think there are any new solutions to it to be found, and I consider every solution I know of ugly. If someone could prove me wrong and provide a clean solution allowing

```
        vector<list<char>> vlc;
```

I would be most happy. However, assuming that does not happen we have a choice between annoying users and introducing some language-technical hack. I have come to believe that the latter is preferable. Many implementors (who?) have – pushed by their users no doubt – have already voted with their feet.

Of the hacks I know of, introducing a state into the lexical analyzer so that >> is interpreted as > > when preceded by two <s at the same bracket nesting level seems the least problematic. However, the exact formulation is not a major concern to me.

10.1 Recommendation

Approve any hack that solves the problem!

11 Acknowledgements

It is had to pin down credit/blame for ideas in a note like this, but I easily spot contributions from Mike Ball, John Barton, Sean Corfield, Andrew Koenig, Lee Nackman, John Spicer, Alex Stepanov, and Erwin Unruh. Parts of this note are borrowed from ‘‘Stroustrup: *Suggestions for Completing Templates* ANSI X3J16/94-0026, ISO WG21/N0413.’’