#X3J16/93-0081, #WG21/N0288

Major Template Issues (Note for discussion)

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This note discusses major unresolved issues related to templates. This includes discussion several extension proposals. Earlier versions of this note has been discussed on the extensions reflector, and I expect that future versions will also be discussed there. That aim of this note is to summarize my understanding of these major issues, and it will be updated as this understanding improves. This note is (necessarily) incomplete. In particular, there isn't yet a detailed discussion of instantiation.

1 Introduction

Please note that this document is not a list of extensions I personally want or have personally suggested. It is a discussion of issues raised over the years in the committee and elsewhere. When I state my personal opinion I do so explicitly, and I have added my opinion of the order of importance of these issues at the end of this note.

Many problems with templates have been pointed out and many changes (extensions, restriction, and alternative resolutions) have been proposed. In particular, papers #92-0133, #92-0014, #93-0007, and #93-0039 deal with template issues.

This note is an attempt to summarize the major issues. I plan to follow up on the individual issues and aim to keep the big picture (the totality of the suggested improvements) in view so that we don't get trapped into dealing with a series of patches each of which solves a problem but fails to take into account related problems.

This note is for discussion, the various ideas have not yet matured into polished proposals for resolutions.

The issues I consider major are:

(F) Function Templates
- Deduction of function template arguments
- Explicit qualification of function template calls
- Overloading of function templates
- Conversions for function template arguments

(C) Class Templates
- Constraints on template arguments
- Overloading of class templates
- Nesting of templates

(B) Name Binding
- Name binding rules for names used in templates
- Name binding and separate compilation

(S) Specialization

(I) Instantiation
- Early instatiation
- Explicit instantiation
- Efficient instantiation

Of these, I consider (B) the hardest followed by (I).

Some of these issues and many issues I consider minor are touched upon in John Spicer's document

#93-0039. A revision of that document (#93-0074) will also be available in the pre-Munich mailing.

2 (F) Function Templates

This section discusses the deduction of template arguments from the argument types in a call, the possibility for explicit specification of template arguments in a call, the current options for overloading a function templates and the possibilities for generalizing this overloading mechanism, and finally the need to allow some conversions for template function arguments.

F1: Deduction of function template arguments

Consider:

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

void g(int* pi, char* pc, int i)
{
        f(pi); // call f(int*), that is f<int>
        f(pc); // call f(char*), that is f<char>
        f(i);  // error: no f(int) declared; that is
               // there is no T for which T* is int
}
```

What are the allowed relations between T and type_expr in

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

where `type_expr` is a declarator?

First, `type_expr` must contain `T`. This is not actually a necessary requirement as I'll discuss in F2 below, but it is a currently accepted rule.

Second, it must be possible to write a call of `f()` that uniquely determines `T` for that call.

Third, a call is legal iff the argument uniquely identifies `T` for that call.

In Portland, we discussed this issue and suggested a set of legal declarators:

```
T
const T
volatile T
T*
T&
T::C    // not C::T
T[n]
C<T>
T (*)(args)
some_type (*) (args_containing_T)
T C::*
C T::*
```

These can be applied in combination. For example:

```
template<class T> f1(T*const);
int *const p;
f1(p);  // T is int

template<class T> f2(const T*);
const int* q;
f2(q);  // T is int
```

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

        template<class T> f3(T::Y*);
        X::Y a;
        f3(&a); // T is X
```

This is ‘‘Other issue #6.3’’ of John Spicer’s document #93-0039.

This is an issue that must be resolved. It is not an extension that we can decide not to consider (though we can decide we don’t want to accept individual deductions). The extension WG voted to recommend the solution in Spicer’s paper unless new problems were discovered. So far, none has been.

We should also accept non-type arguments for template functions (as requested by the libraries WG). For example:

```
        template<int i> void f(Bits<i>&);

        void g(Bits<10>& b)
        {
                f(b); // i is 10: f<10>(b)
        }
```

Like a type argument, a non-type argument is acceptable only if it can be deduced from the type an actual argument. This means that it must appear as a class template argument or as an array bound. Thus we can add

```
                C<i>
                some_type[i]
```

to the list of acceptable declarators above.

F2: Explicit qualification of function template calls

Consider:

```
        template<class T> T create();
```

This declaration is currently illegal because the argument `T` cannot be deduced from a call of `create()`. This could be allowed if function template arguments could be specified explicitly in a call:

```
        void g()
        {
                create<int>();   // T is int
                create<char*>(); // T is char*
        }
```

This was discussed in my original 1988 template paper and rejected because I couldn’t see how to handle this ambiguity:

```
        void g()
        {
                f<1>(0); // (f) < (1>(0)) or (f<1>) (0) ?
        }
```

I now don’t consider this a real problem. If `f` is a template name `f<` is the beginning of a qualified template name and the tokens following must be interpreted based on that; if not, `<` means less-than. We can’t parse C++ without that rule.

Consider:

```
template<class T, class U> T convert(U);

void g(int i)
{
        convert(i);  // error template argument T not specified

        convert<char*>(i);  // T is char*, U is int

        convert<char*,double>(i); // ?
}
```

We have alternatives for the last example.

I suggest that we believe the explicit qualification (that is, `U` is `double`) and convert the actual argument accordingly. Thus we get

```
        convert<char*,double>(i); // convert<char*>(double(i))
```

This is simple and also gives the programmer an explicit mechanism where type deduction seems too complicated. The alternative of believing the type deduction and ignoring the explicit qualification would be perverse. The alternative of requiring the type deduced to exactly match the explicit qualification seems pedantic and useless.

Suggested rules:

(1) If function template arguments are specified in a call they are specified in the order they were declared as template arguments. Trailing template arguments may be omitted.

(2) Any template argument not explicitly specified must be deduced according to the rules in F1.

For example:

```
template<class T1, class T2, class T3> T1 f(T2,T3,int);

void g()
{
        f<int,int,int>(1,2,3);
        f<int,int>(1,2,3);      // T3 deduced to be int
        f<int,,int>(1,2,3);     // syntax error
        f<int>(1,2,3);          //  T2 and T3 deduced to be int
        f<>(1,2,3);             //  error T1 can't be deduced
        f(1,2,3);               //  error T1 can't be deduced
}
```

We could outlaw the notation `f<>` but unless someone has a good reason not to allow it I think we should accept it.

Consider also:

```
        template<class T1, class T2> void f(T2);
```

This is currently illegal because there is no way of deducing `T1`. The proposed rules allow it and uses have been suggested long before this particular mechanism was invented:

```
template<class T1, class T2> void f(T2 a)
{
        int i = 0;
        // ...
        T1 temp = a[i]; // use T1 to control precision of computation
        // ...
}

void g(Array<float>& a)
{
        f<float>(a);
        f<double>(a);
        f<Quad>(a);
}
```

Explicit specification of template arguments is an extension that we can decide not to deal with. I consider it a simple and useful generalization that we ought to accept.

The main problem with this proposal is that it does not appear to fit naturally into type-safe linkage schemes that don’t take the return type into account. For example:

```
template<class T> T create();
```

allows us to use functions

```
void g()
{
        int i = create<int>();
        double d = create<double>();
}
```

How would this be fitted into an existing linkage scheme? My first answer is that it is obvious from the template declaration that the functions generated from this template don’t fit into an existing scheme that doesn’t take the return type into account. We can therefore extend such schemes compatibly. Consider a name encoding scheme like the one described in the ARM (that is, basically Cfront’s). We simply add a character meaning "return type follows," say ‘r’, and we get

```
void g()
{
        int i = create<int>();          // create__FVrI
        double d = create<double>();    // create__FVrD
}
```

This would imply that you could specialize `create()` only if you had a declaration of the function template in scope:

```
float create() { ... }  // no template in scope
                        // => linkage name create__FV

template<class T> T create();

char* create() { ... }  // linkage name: create__FVrPC
```

There are other reasons for requiring a template declaration in scope when a specialization is declared; see (S).

As ever, providing several functions with the same name, same argument types, and different return types should cause compiler or linker errors (though some return type errors excape detection in some current implementations).

F3: Overloading of function templates

Type deduction (as in F1) is easy provided no an exact type match is required for actual arguments and and provided no overloading is allowed. That is, the reason I prohibited template overloading and conversion of template argument in the ARM. For example:

```
template<class T> void f(T*);
template<class T> void f(T&);  // error: can’t overload template

template<class T> class D : public B<T> { ... };

template<class T> g(B<T>* pb);

void h(D<int>* pd)
{
        g(pd);  // error not a match for  g(B<int>*)
                // even though D<int> is derived from B<int>
}
```

Naturally both decisions have been considered constraining.

Function templates actually permit two forms of overloading: Overloading by ordinary (non-template) functions and specialization. For example;

```
        template<class T> void f(T);
        void f(int);     // specialization where T is int
        void f(int,int); // overloading
```

The first observation is that this implies that we already has the

```
        void g()
        {
                f<1>(0); // (f) < (1>(0)) or (f<1>) (0) ?
        }
```

problem, after all if we consider f to be the overloaded function we choose the

```
                (f) < (1>(0))
```

resolution, and if we consider f to be the template name we choose the

```
                (f<1>) (0)
```

resolution. We currently choose the latter (as I understand things, we need an explicit resolution on this; see John Spicer’s paper), and I don’t propose to change that.

The second observation is that we have a rather specialized and tricky mechanism. Could we simplify it by generalizing it to include overloading by template functions?

Observation: We could allow template argument types where the sets of acceptable argument types are disjoint:

```
        template<class T> void f1(T);
        template<class T> void f1(T, int);

        void g1()
        {
                f1(1);    // must be f1<int>(int)
                f1(1,1);  // must be f1<int>(int,int)
        }

        template<class T> void f2(T*);
        template<class T> void f2(Vector<T>&);

        void g2(int i,Vector<int>& v)
        {
                f2(&i);   // must be f2<int>(int*)
                f2(v);    // must be f2<int>(Vector<int>&)
        }
```

The problem comes where the sets aren’t disjoint:

```
        template<class T> void f3(T*);
        template<class T> void f3(T);

        void g(int i)
        {
                f3(&i);  // f3<int>(int*) or f3<int*>(int*) ?
        }
```

If template function overloading were allowed we could either disallow this call as ambiguous or choose to call

```
                f3(&i);  // f3<int>(int*)
```

because the match of `int*` for `T*` is more specific than the match of `int*` to `T`. Alternatively we could disallow the declarations

```
template<class T> void f3(T*);
template<class T> void f3(T);
```

because of the potential ambiguity and only allow overloading by template functions with disjoint argument type sets such as `f1` and `f2` above.

Unless we allow the most general case (allow even `f3`) people will complain. However, general overloading combined with general conversions gets very complicated, probably even undecidable in some cases, so I'll not even discuss anything but the simplest variations.

Consider also this example:

```
template<class T> void f4(const T*);
template<class T> void f4(T*);

void g(int i, const int* p)
{
        f4(&i); // f4<int>(int*)
        f4(p);  // f4<int>(const int*)
}
```

One issue that one must consider is how to represent the different functions generated or explicitly declared by users. As long as only one function can be generated with a given list of function argument types the currently used techniques for distinguishing overloaded functions to the linker can be used. If that is not so something further is needed. Consider:

```
// file 1:
        template<class T> void f3(T*);

        void g(int i)
        {
                f3(&i);  // f3<int>(int*)
        }

// file 2:
        template<class T> void f3(T);

        void g(int i)
        {
                f3(&i);  // f3<int*>(int*)
        }
```

Thus we can either accept unsafe linkage or enhance our linkage representation. Note that this last example can be written now and the error (relative to current rules) can't be detected in a single compilation. How do the current implementations handle this example? This is not a rhetorical question, please try and post results; Cfront causes this error to be caught by the linker.

Note that because of specializations we can't encode templates differently from non-templates

```
// file 1:
        template<class T> void f3(T*);

        void g(int i)
        {
                f3(&i);  // f3<int>(int*)
        }

// file 2:
        void f3(int* p) { /* ... */ } // defines f3<int>(int*)
```

I'll discuss specializations and the problems they cause in (S) below.

Note that changes that affect linkage are the hardest to deploy in real implementations for real users.

I am in favor of looking at template function overloading in detail and I think it is feasible and that it could be useful. I don't yet have an opinion of whether it is worthwhile or not. Note that we already have overloading of template functions with non-template functions. If we extend those rules we'd have to refine the ambiguity resolution rules. See also (F3) below.

If we allow overloading of template functions we encounter a problem with explicit specification of template arguments. Consider:

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

        void g()
        {
                f<double>(1,1);  // f<double,double> or f<doubl,int> ?
        }
```

When do we do overload resolution? We first fix the template argument(s) as explicitly specified, and then do resolution (according to whatever rules we choose on the remaining function arguments).

(F3) Conversions for function template arguments
Naturally, people have found examples where they considered conversions of values used as template arguments essential. The ARM prohibits such conversions.

Consider:

```
        template<class T> class B { /* ... */ };
        template<class T> class D : public B<T> { /* ... */ };

        template<class T> f(B<T>&);

        void g(B<int> bi, D<int> di)
        {
                f(bi);  // ok: exact match
                f(di);  // ??: di is a D<int> and therefore also a B<int>
        }
```

The ARM says `f(di)` is an error, yet it seems reasonable to support it. In fact, the implementor of template functions in Cfront (not me for a change :-) considered it so essential that he added it as an extension. Similar for:

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

        int a[10];
        const int i = 12;

        void g()
        {
                min(a,i);
        }
```

The second argument isn't an `int` (it's a `const int`) and is therefore outlawed by the exact type match requirement in the ARM. However, such trivial conversions do seem harmless and useful.

So, if we allow conversions, which conversions can we allow without getting into new trouble?

Consider allowing only the following:

– `D<T>*` to `B<T>*` where `B<T>` is an accessible base of `D<T>`.
– `D<T>&` to `B<T>&` where `B<T>` is an accessible base of `D<T>`.
– `x T` to `y T` where `x` and `y` are `const`, `volatile`, or empty.

If we allowed overloading we would have to deal with

```
template<class T> f(B<T>&);
template<class T> f(D<T>&);

void g(B<int> bi, D<int> di)
{
        f(bi);  // ok: exact match
        f(di);  // ok: exact match
}
```

so we'd need a (hopefully simple) form of the overload matching rules. John Spicer presents the minimal extension for that in his upcoming revision of his list (#93-0039).

I suspect we will have to accept conversions at least to the extent Cfront (and other implementations) do. The absence of such simple conversions does cause inconvenience and confusion.

3 (C) Class templates

This section discusses the possibility of specifying constraints for template arguments, the possibility of overloading class templates, and finally the possibility of allowing nested templates. Some of these issues generalizes to function templates, but are best first discussed in the context of class templates.

(C1) Constraints on template arguments

One of the most frequently heard comments about templates is that template arguments are unconstrained and that all kinds of benefits would be achieved if they were ''properly constrained.'' Curiously enough, I agree and even discussed the issue in my original 1988 paper. However, I was stumped by the problem of how to express constraints.

Consider people's favorite constraint ''the `T` must be a type derived from `B`:''

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

That's easy to represent and easy to implement. However, I'm strongly against any constraint system that provides constraints exclusively as requirements based on derivation. In my opinion, it will bias system design towards lots of little base classes that are basically noun forms of verbs; that is, individual functions turned into classes. People won't be able to say ''`T` must have an `f()`'' and will say ''`T` must be derived from `F`'' where `F` turns out to be something like

```
class F {
public:
        static void f();
};
```

That is often indirect, obscure, verbose, and inefficient. There are cases where this style is ideal (that is, the most elegant, least verbose, and most efficient solution), but they are not the majority and we mustn't legislate exclusively in favor of them.

Consider other constraints:

```
template <class T : *>  // T must be a pointer
template <class T : &>  // T must be a reference
template <class T : *[10]> // T must be a pointer to an array of 10 elements

template <class T : B,C> // T must be derived from B and C

template <class T : { +, =, f, T(int) }>
        // T has +, =, f, and a constructor from int

template <class T : { ::+, =, f, ::g }>
        // T uses global + and g, and has =, f

template <class T : B { +, =, f, T(int) }>
        // T must be derived from B and has ...

template <class T : * { ::+, =, f, ::g }>
        // T must be a pointer to something that has ...
```

The syntax is unique (bad) and doesn't express exact signatures but rather specifies names used. I think the latter may be an advantage; see (B) below. Anyway, specifying complete signatures would be verbose.

I once solved the constraint problem for my programs by taking advantage of a local implementation restriction: Cfront does a complete syntax and semantic check of all inline functions at the point where a template declaration is instantiated. For example:

```
template<class T> class X {
        void constraints_check(T* tp)
        {
                B* bp = tp; // T must have an accessible base B
                tp->f();    // T must have a member function f
                T a(1);     // T must have a construct accepting an int
                a = *tp;    // T must have assignment
                // ...
        }
        // ...
};
```

This allowed me to express any constraint and have it conveniently checked. I even found that if I chose the local variable names well I could tailor my error messages to be very meaningful (better than most of the ones the compiler composes).

The only snag is that we can't require inline functions to be checked unless they are used. Well, we could, but I think that would be a mistake.

However, we here have a way of expressing arbitrary constraints in what I think of as a very convenient way. All we have to do is to find a way to tell the compiler to do it for us. Every compiler already knows how.

First idea: add a keyword:

```
template<class T> class X {
        constraint void dummy(T* tp) // new kind of function
        {
                T* tp = 0;
                B* bp = tp; // T must have an accessible base B
                tp->f();    // T must have a member function f
                T a(1);     // T must have a construct that accepts an int
                a = *tp;    // T must have assignment
                // ...
        }
        // ...
};
```

That is, a ''constraint'' is an inline function that must be fully checked an the point where a template declaration is generated.

There is a multitude of alternatives. For example:

```
template<class T>
        constraint      // new syntax
        {
                T* tp = 0;
                B* bp = tp; // T must have an accessible base B
                tp->f();    // T must have a member function f
                T a(1);     // T must have a construct that accepts an int
                a = *tp;    // T must have assignment
                // ...
        }
        class X {
                // ...
        };
```

That is, a ''constraint'' is an unexecutable piece of code that must be type checked when a template class is generated for a set of template arguments.

I offer no prices for the longest list of alternatives. In the following, I'll use this last syntax because it generalizes to expressing constraints for function templates.

One could consider a simpler syntax for simpler constraints but I suspect that would be a mistake because that would add complexity and encourage the use of the simple constraints that would often enough be less than ideal.

The constraints that it is least obvious to express this way are two of the simplest ones ''`T` must be a pointer'' and ''`T` must be a reference.'' The pointer constraint is actually easy:

```
template<class T>
        constraint {    // T is a pointer
                T pt = 0;
                const void* pv = pt;
        }
```

but I have not found a way of expressing the reference constraint:

```
template<class T>
        constraint {    // T is a reference
                // what might go here?
        }
```

Nor do I expect to find one. I found that I missed being able to say ''`T` is a reference'' and also ''`T` is not a pointer. We could probably do with only one of those, but I suspect that if we want constraints we'll eventually have to adopt pseudo-functions known to the compiler:

```
template<class T>
        constraint {
                is_reference(T);
        }
```

and

```
template<class T>
        constraint {
                not_pointer(T);
        }
```

This is getting complicated! Can you think of any other constraint that cannot be expressed through a simple use? Is any such constraint useful?

If we had constraints would they be useful for template functions also? The answer is less obvious because template argument deduction serves a similar purpose. For example:

```
template<class T> void f(T&);  // argument must be a reference
template<class T> void f(T*);  // argument must be a pointer
```

However, the set of things expressed by constraints and through function argument types are independent.

For example, here is a function `f()` that takes a pointer to a type that has a member `f()` and an output operation:

```
        template<class T>
                constraint {
                        T* pt = 0;
                        pt->f();        // T must have a member f
                        cout << *pt;    // T can be output
                }
                void f(T*);     // argument must be a pointer
```

I suspect that if we have constraints for class templates they will be requested for function templates. Also, as shown, constraints and template argument deduction have different properties (because their primary aims are different).

As stated here, constraints would be compiled/checked in the scope surrounding the function or class template that they were attached to. This would currently be the global scope (because templates must be global). However, that might change; see (C).

(C2) Allow overloading of class templates

If we have constraints in a suitable form we might overload class templates based on them. For example:

```
        class D : public B { ... };
        class E : public C { ... };

        template<class T : B> class X { ... };  // XB
        template<class T : C> class X { ... };  // XC

        D d;
        E e;

        X<d> x1;        // clearly XB
        X<e> x2;        // clearly XC
```

If constraints are expressed through use as above we get the same result:

```
        template<class T>
                constraint { B* pb = (T*)0; }
                        class X { ... };  // XB
        template<class T>
                constraint { C* pb = (T*)0; }
                        class X { ... };  // XC
```

(sorry about the casts).

```
        D d;
        E e;

        X<d> x1;  // XB
        X<e> x2;  // XC
```

in this case we'd have to evaluate the constraint for both class templates and in each case pick the one that worked.

There seems to be two choices for dealing with ambiguities arising from overloading: Accept overloading if and only if there is exactly one match or invent some best match strategy. If constraints are expressed through use then I think that ‘‘exactly one match’’ is the only strategy, if constraints are based of some type specification a best match strategy like the one suggested for function template arguments on F3 becomes possible.

I am personally not very keen on class template overloading but would be willing to accept a scheme based on ‘‘exactly one match.’’ Additionally, I'd accept a scheme based on different number of arguments:

```
template<class T, int i> Array { ... }; // 1D
template<class T, int i, int j> Array { ... }; // 2D

Array<int,10> a; // 1D
Array<int,10,10> b; // 2D
```

rather than the current:

```
template<class T, int i> Array1D { ... };
template<class T, int i, int j> Array2D { ... };

Array1D<int,10>
Array2D<int,10,10>
```

One might consider overloading class templates based on the argument types of their constructors. For example:

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

template<class T> class X {
        // ...
        X(T,T);
        // ...
};

X<int> a(1);    // first X<int>
X<int> a(1,2);  // second X<int>
```

However, since there a class may have several constructors and that constructors of any argument type can be defined for a class allowing that seems to be very unstructured and error-prone.

(C) Allow nesting of templates
<<I'll save this discussion for later, I don't think any fundamental design issues are involved, i.e. I don't think the issues are that major>>

4 (B) Name binding rules

So much for the easy problems, the real problem, and one that we *must* solve because it is a necessary resolution rather than an extension is to specify exactly the rules for binding names used in a template to their declarations.

The fundamental problem was described by Martin O'Riorden in #92-0133.

I conjecture that if we can solve this particular example to our satisfaction then the rest will fall in place:

```
template<class T> T sum(Vector<T>& v)
{
        T t = 0;
        for (int i = 0; v.size()<=i; i++)
                t = t + v[i];
        if (DEBUG) cout << "sum is " << t << '0;
        f(t);
        f(i);
        return t;
}
```

```
        // define a a type C

        void f(Vector<C>& v)
        {
                C c = sum(v);
        }
```

I deliberately didn't use t+=v[i] because I want to discuss + and = separately. The calls of f() are there simply to allow me to elaborate a point later on.

I will deal with the cases where the template definition isn't in the same compilation unit in (B2) below.

Note: The Vector template must be defined and have a size() member function. The use of a defined template causes no problems and is equivalent to the use of a built-in type in this context. Since v is of a class that has T as a template argument the result of a function depending on v can possibly be of type T or a type derived from T; only by examining the definition of Vector can we know.

Whichever type is used for T must have the following properties:

– A T can be initialized by 0.
– Two Ts can be added by +.
– T has assignment.
– The sum of two Ts must be of a type that can be assigned to a T (note T+T need not be a T).
– There is a global function f() that can be called with a T argument.

There are several different ways of fulfilling these requirements:

(1) A built-in type, such as int, plus a function f(int) has the properties.

(2) A class C1 where all operations are member functions:

```
        class C1 {
                // ...
                C1 operator+(const C1&);
                C1& operator=(const C1&);
                C1(int);
        };

        void f(const C1&);
```

has the properties.

(3) A class C2 using a combination of member function and global functions:

```
        class C2 {
                // ...
                friend C2 operator+(C2,C2);
                friend void f(C2);
                // default operator =
                C2(int);
        };
```

has the properties (this is how I usually define class complex).

The signatures of C1 and C2 are different and I could have conjured up even more different signatures by relying on user-defined conversions, etc. The point is that it does not seem reasonable to require the designer of the sum() template to specify the exact signatures of the required T functions: there are simply too many reasonable alternatives (for example, built-in operations, global functions, member functions, reference arguments, value arguments). Picking one alternative would make sum() less useful.

Note that the global function operator+(C2,C2) need not have been a friend and might therefore never have been mentioned in the declaration of C2.

When writing a template the programmer can usually separate the non-local names into two categories:

(N1) Names related to the template arguments (for example, T(int), +, =, <<, size, and maybe <=, f(T)).

(N2) Global names not related to template arguments and in scope at the point of the template definition (cout, DEBUG, <<, f(int)).

One problem is that only the programmer knows which is supposed to be which.

I think that we can agree that names not in N1 or N2 – that is, picked up from the environment of the use of a template – is undesirable ‘‘crud.’’ Picking up such crud would give templates an undesirable macro flavor. At least some current implementations can be fooled into picking up such crud, but that problem can and should be fixed.

Paper #92-0133 argues (convincingly, I think) that the global names not related to the template arguments must be bound at the point of the template definition. I think the problem with #92-0133 is that it doesn’t tell us how to recognize the difference between (N1) and (N2). The `C1`, `C2` examples shows that we can’t just say that all global names must be bound at the point of the template definition because we

– don’t know which names are global, and
– some global names related to the template argument are not yet available.

Could we demand that that all global names were somehow known at the point of the template definition? That is, could we give the programmer a way of distinguishing the (N1) and (N2) categories? For example;

```
template<class T> T sum(Vector<T>& v)
        #global: DEBUG, cout, <<(char*)
        #related: T(int),+(T,T),<=(maybeT,int),
                  =(T,+(T,T)),<<(T),<<(maybeT,char)
{
        T t = 0;
        for (int i = 0; v.size()<=i; i++)
                t = t + v[i];
        if (DEBUG) cout << "sum is " << t << ’0;
        f(t);
        f(i);
        return t;
}
```

Meaning that the `#global` names are looked up immediately and the lookup of `#related` is done once a particular `T` is known.

This is far too detailed, clumsy, elaborate, verbose, etc., for people to accept (I think), and too hard to get right.

Why would people want to distinguish the categories? Primarily to ensure that the compiler doesn’t resolve the calls to some unrelated function declared after the template and before the argument type. Disallowing this third possible source of declarations also simplifies the definition of what the context of a template instantiation is and thereby simplifies enforcement of the one definition rule and ought to speed up template instantiation.

Simple rule:

(1) When compiling the template look up all potentially global names (including operators) and remember what they bound to if found. Give an error if a name that is not a function (or an operator) isn’t found.

(2) look up all names at the point where the template arguments are specified. Give an error if a name is not found in (1) or (2). Give an error if a name found in (2) doesn’t name a function (or an operator) dependent on a template argument. If a name is found in both (1) and (2) use the one found in (2) (i.e. the one that depends on the template argument).

I will define ‘‘point where the template arguments specified’’ and ‘‘depend on a template argument’’ below.

I expect ‘‘name found in both (1) and (2)’’ might be a case for a warning if the dependency on a template argument is non-obvious.

One might consider being more specific. For example, one might give an error unless a global name is declared at the point of use in the template definition whether dependent on a template argument or not. This would force the user to write something like:

```
#include <iostream.h>

template<class T> T sum(Vector<T>& v)
{
        extern int DBUG;
        extern void f(int);
        extern void f(T);

        T t = 0;
        for (int i = 0; v.size()<=i; i++)
                t = t + v[i];
        if (DEBUG) cout << "sum is " << t << ’0;
        f(t);
        f(i);
        return t;
}
```

The problem here is that the writer of `sum()` would have to make assumptions about the argument type of `f()` and that isn’t reasonable. There are too many alternatives:

```
        void f(T);
        void f(T&);
        void f(const T&);
        void f(TT); // there is a conversion from T to TT
        ...
```

This is the reason I don’t propose a notation for saying ‘‘this function somehow depends on `T`’’ in the template definition.

I thought of adding a way of letting the writer of the argument type say ‘‘this ‘‘this function somehow depends on me.’’ A friend function says something like that:

```
class C2 {
        // ...
        friend C2 operator+(C2,C2);
        friend void f(C2);
        // default operator =
        C2(int);
};
```

However, it is not reasonable to require all dependent functions to be friends because we also wants to minimize the set of functions we grant friendship to. For example, I prefer this version to the one above:

```
class C2 {
        // ...
        C2 operator+=(C2);
        friend void f(C2);
        // default operator =
        C2(int);
};

C2 operator+(C2 a, C2 b) { C2 t = a; return t+= b; }
```

It also seems wrong to require the writer of an argument type to conform to requirements of unknown users.

However, please note that a user can explicitly declare non-dependent names within the template. For example:

```
template<class T> T sum(Vector<T>& v)
{
        extern int DBUG;
        extern void f(int);
        extern void f(T);

        // ...
```

We just can't make it compulsory.

The ''point where the template arguments are specified'' is the point where names in the template definition that are dependent on template arguments are bound. This point, let's call it the binding point, is immediately before the global declaration that first names a template class or template function for a given set of template arguments. This is not intended to be a new definition. You can find it in the ARM. For example:

```
// binding point for for Vector<Complex>
Vector<Complex> d;

// binding point for sum(Vector<Complex>&)
void f()
{
        Complex c = sum(d);
}
```

The purpose of moving the binding point to the global level is to avoid picking up local names (for example, `c()` or `d()`).

There are thus exactly two points of interest for determining the meaning of a template definition: the point of the template definition itself and the point of binding for a given set of template arguments.

Note that if two points of binding for the same template given the same template argument in different compilation units yields different name bindings then the one definition rule has been violated. We can therefore talk as if there is only one point of binding and consider the problem of detecting violations of the ODR separately.

A name is dependent on a template argument `T` iff:

(1) it is a member of `T`, or

(2) takes an argument that determines `T` according to the template argument deduction rules from F1.

Note that the suggested lookup rule requires that every name that is supposed to be a type name is explicitly specified to be a type name at the point of the template definition. This, I consider important and desirable. For example:

```
template<class T> TT sum(Vector<T>& v)  // error TT not defined.
{
        TT t = 0;       // error TT not defined
        for (int i = 0; v.size()<=i; i++)
                t = t + v[i];
        if (DEBUG) cout << "sum is " << t << '0;
        f(t);
        f(i);
        return t;
}
```

This proposed resolution of the name lookup/binding issues builds directly on the rules in the ARM, the discussion in #92-0133, and requires no extensions. It disallows most of the known examples of a name being ''hijacked'' by a context of an argument type (that is, their resolution takes away most of the ''macro flavor'' left in templates), and allows for early detection of many errors in a template definition. The resolution allows complete syntax check of template definitions at the point of their definition. The resolution leaves all of the examples of published templates I have looked at legal (I haven't looked very carefully, though). The resolution allows templates with unexpected, but reasonable context dependencies to be repaired by adding declarations of global names at the point of the template definition. The resolution simplifies the implementation of templates by reducing their context dependency.

Examples (based on the ones Erwin Unruh posted). ''lookup 1'' is the name lookup at the point of the template definition, ''lookup 2'' is the name lookup at the point of binding.

First consider names that are not dependent of the template argument:

```
class P;

int a();

template < class T > int temp ( T t )
{
        extern int b();

        a();    // found in lookup 1
        b();    // found in lookup 1
        c();    // found in lookup 2    // error, not dependent
        d();    // not found            // error
        e();    // not found            // error
}

int c();

void x()
{
        extern int d();
        temp(P);
}

int e();
```

Note that `c()` cannot be referred to even though it was in scope for lookup2 because it isn't dependent; `d()` isn't found because the point of binding is before the local definition.

Now consider dependent names:

```
class P;

int a(P);

template < class T > int temp ( T t )
{
        extern int b(T);

        a(t);   // found in lookup 1 and 2
        b(t);   // found in lookup 1 and 2
        c(t);   // found in lookup 2    // ok, dependent
        d(t);   // not found            // error
        e(t);   // not found            // error
}

int c(t);

void x()
{
        extern int d(t);
        temp(P);
}

int e();
```

Now consider overloading:

```
class P;

int a();

template < class T > int temp ( T t )
{
        extern int b();

        a(t);   // found in lookup 1 and 2
        b(t);   // found in lookup 1 and 2
        c(t);   // found in lookup 2    // ok, dependent
        d(t);   // not found            // error
        e(t);   // not found            // error
}

int c(t);

int a(T);
int b(T);

void x()
{
        extern int d(t);
        temp(P);
}

int e();
```

Here, the calls of `a()` and `b()` leads to a rebinding to the dependent functions `a(T)` and `b(T)`. Such rebinding seems a plausible candidate for ‘‘suspicious call’’ warnings, but we’ll need experience with actual code to be sure such a warning wasn’t prompted by legitimate code.

(B1) Name binding and separate compilation

The basic idea is that the meaning of a program should be independent of how it is composed of separately compiled units. This ideal isn’t perfectly achievable, but in the proposed name binding rules approximate this ideal.

Consider the way I’d really write the `sum()` example:

```
// file sum.h:
        template<class T> T sum(Vector<T>& v);

// file sum.c:

        #include<sum.h>
        #include<iostream.h>

        // ...

        extern int DBUG;
        extern f(int);
        template<class T> T sum(Vector<T>& v)
        {
                T t = 0;
                for (int i = 0; v.size()<=i; i++)
                        t = t + v[i];
                if (DEBUG) cout << "sum is " << t << ’0;
                f(t);
                f(i);
                return t;
        }

// file C.h:

        // ...

        class C {
                // ...
                C operator+=(C);
                friend void f(C);
                // default operator =
                C(int);
        };

        inline C operator+(C a, C b) { C t = a; return t+= b; }

        // ...

// file user.c:

        // lots of distracting stuff 1

        #include<sum.h>
        #include<C.h>

        // lots of distracting stuff 2

        void f(Vector<C>& v)
        {
                C c = sum(v);
        }
```

When `sum.c` is compiled complete syntax checking can be done and the compiler can verify that there are no undefined variables and types. It may also record bindings of `f`, `DBUG`, and `cout` for future reference (on the other hand, it need not do so).

In principle, `C.h` could also be syntax checked and many name bindings recorded. Alternative, it can simply be included textually where needed in the traditional fashion.

When `sum.c` is compiled we have information enough type check everything and to tell an ‘instantiator’ (or whatever the part of the compilation system that makes versions of a template for a specific template arguments is called) that `sum(vector<C>&)` is needed.

A simple instantiator could simply produce `sum(vector<C>&)` by including `sum.c` with the template argument replaced by `C` in `user.c` at `sum(Vector<C>&)`’s point of binding getting:

```
// instantiator’s file:

        // lots of distracting stuff 1

        template<class T> T sum(Vector<T>& v);
        class C {
                // ...
                C operator+=(C);
                friend void f(C);
                // default operator =
                C(int);
        };

        inline C operator+(C a, C b) { C t = a; return t+= b; }

        // ...


        // lots of distracting stuff 2

        // point of binding for sum(Vector<C>&)

        #include<iostream.h>

        // ...

        extern int DBUG;
        extern f(int);

        C sum(Vector<C>& v)
        {
                C t = 0;
                for (int i = 0; v.size()<=i; i++)
                        t = t + v[i];
                if (DEBUG) cout << "sum is " << t << ’0;
                f(t);
                f(i);
                return t;
        }

        // ...

        void f(Vector<C>& v)
        {
                C c = sum(v);
        }
```

To generate the right code, the instantiator must avoid changing the meaning of `sum(Vector<C>&)` by binding names to what I have labeled ‘‘distracting stuff.’’ This can be done my simply ignoring any information not avialable in the original `sum.c`. If the instantiator isn’t smart enough it might fail to compile the example (for example, if the ‘‘distracting stuff’’ messes up the context so badly that the code cannot be parsed, but it will not be too hard to ensure that if code is generated then the binding will be those defined by the lookup algorithm described above.

An instantiator relying on partially compiled stuff can handle things more elegantly: It can simply know which global names were available to `sum.c` and ignore the ‘‘distracting stuff’’ completely.

What if there are functions depending on `T` among the ‘‘distracting stuff.’’ Unfortunately, I don’t see any way of avoiding picking that up. There is no language mechanism that allows us to distinguish a function depending on `C` that was meant to be used by the designer of `C` from one that was simply supplied in a local environment. I would seriously consider issuing a warning if a function in a template was bound to a function dependent on `C` that wasn’t declared in the file where `C` was declared (say, `C.h`). This problem comes from deducing what is ‘‘dependent on `T`’’ rather than explicitly declaring what is ‘‘dependent on `T`.’’

What if `sum(Vector<C>&)` is used in two different compilation units each with its own point of binding? If only names from `sum.c` and `C.h` are used in the bindings there will be no difference between the two functions generated (and we can thus do the compilation once only). If both compilations of `sum(Vector<C>&)` pick up the very same dependent functions from the ‘‘distracting stuff.’’ Unfortunately, this unlikely occurrence must be legal. If, however, different dependent functions from the ‘‘distracting stuff’’ are used then the one definition rule has been broken.

In principle, this can all be checked. In practice, it can be expensive to check. It is very tempting to bless the idea of totally ignoring the ‘‘distracting stuff’’ and compiling `sum(Vector<C>)` based exclusively on `sum.c` and `C.h`. I am against requiring a user to specify that a template should be generated for a particular set of template arguments; I'm less against the idea of allowing a user (optionally) to do so. For example:

```
#include<sum.h>
#include<C.h>
generate sum(Vector<C>&);
```

Can anyone think of a formulation that would allow us to specify something roughly equivalent to this last suggestion? This will be discussed further in (I) below.

5 (S) Specialization

Specialization is a form of overloading intended exclusively to allow the programmer to provide alternative implementations for specific template arguments. It is a very restricted and rather unconstrained facility.

The issue is to see whether it is possible to make specialization ‘‘better behaved’’ and fit into a more general scheme of overloading without breaking compatibility where compatibility matters and without imposing restrictions that would damage efficiency where that matters.

Consider:

```
// file1:

        template<class T> T sum(Vector<T>& v) { /* ... */ };

        // ...

        Vector<int> vi(100);

        int s = sum(vi);
```

seems simple and obvious. However:

```
// file2:

        int sum(Vector<int>& v) { /* ... */ };
```

may specify a completely different meaning for `sum()` so that the reader of `file1` cannot make any assumptions about what is going on without examining every other source program file to see if there is a specialization like the one in `file2`.

This differs from ordinary overloading in that there really are two definitions of `sum(Vector<int>&)` plus a rule that gives priority to one.

Worse, specialization can't just change semantics, it can also change name binding and (therefore) parsing:

```
        int x;

        class MM { ... };

        template<class T> class X { public int x; };

        template<class T> class XX : public X<T> {
                void f() { MM a; x++; }
        };
```

It wouldn't be unreasonable for the writer of `XX` to assume that `MM` was the class `MM` declared above and that the `x` incremented was `X<T>::x`. However, given unconstrained specialization this cannot be assumed!:

```
        class X<int> { class MM { ... }; /* no x */ };
```

Thus `XX<int>` will find itself using `X<int>::MM` rather than `::MM` and `::x` rather than the non-existent `X<int>::x`.

For better or worse, the class `XX<int>` can be generated. It is easy to construct a specialization that causes syntax errors:

```
        class X<double> { int MM; typedef int x; };
```

Now `XX<double>::f()` is defined as

```
        void XX<double>::f() { X<double>::MM a; X<double>::x++; }
```

which contains two syntax errors. The problem is that a specialization of a base class may or may not intercept the same set of names as its general version.

This is unacceptable. It is acceptable that one function name can denote different functions for different argument types, but not that the same names in a class declaration can mean different things dependent on subsequent declarations. At least, it must be required that a class template cannot be specialized after a use that requires the class declaration to be known - such as a use as a base class (this requirement was suggested in #92-0133).

Where can we draw the line between "acceptable" specializations and unacceptable ones? Can we draw such a line? I think we must. How can we ensure that the rule drawing the line is enforced?

Suggestion 1: Require the template declaration to be in scope at the point of a specialization definition (or else the compiler may assume that a function defined isn't a specialization of a template).

This would allow template functions to use a different representation from other functions (but does not require it to; link compatibility can be maintained). It would allow compiler to enforce whatever rules for correspondence between a template and its specializations we might decide on and enable warnings even if we don't. It would allow us to minimize the effect of ‘‘hijacking’’ templates through specialization by allowing us to find the set of files where a specialization might occur.

Suggestion 2: Require a declaration of a specialization to be in scope at the point of use (or else the compiler may assume that there is no specialization).

For example:

```
        template<class T> T sum(Vector<T>& v) { /* ... */ };

        // ...

        Vector<int> vi(100);

        int s = sum(vi); // can assume sum() as defined above
                         // is used (we can't under the current rules)

        int sum(Vector<int>& v) { /* ... */ };  // error: specialization
                                                // after use
                                                // (no error under the current rules)
```

This would lead to a minor problem with overload resolution because the user would have to write:

```
template<class T> T sum(Vector<T>& v) { /* ... */ };
int sum(Vector<int>& v);

// ...

Vector<int> vi(100);

int s = sum(vi); // calls sum(Vector<int>&)
```

and an explicit declaration takes priority over a template declaration under the current overload resolution rules. However, I don't consider this a serious problem.

It would also outlaw the defining a specialization in another source file without declaring the intent to do so.

These suggestions makes specialization more constrained and more predictable. What concrete benefits can we expect from that?

6 (I) Instantiation

<<here I'll discuss early instantiation, explicit instantiation, efficient instantiation, and enforcement of the ODR>>

Explicit Instantiation

Why don't we have an instantiate operator? For example (as suggested in (B) above):

```
#include<sum.h>
#include<C.h>
generate sum(Vector<C>&);
```

My reason for not including an instantiate operator in C++ was that it would put the burden of instantiation on the individual programmer. Because most of the templates that need instantiation will be defined in libraries this is exactly the wrong place to put the burden. The user doesn't know what templates are actually used in a program because libraries themselves often use templates. Thus the instantiation of one template can cause the instantiation of other templates – often templates internal to a library that the user might never have heard of. On the other hand, the library don't know which instantiations will be needed by users.

Also, the use of an instantiation operator twice for the same template with the same argument would be an error. Thus if you compose software out of separately developed parts, each of which use the instantiate operator, there is a real chance that multiple uses of the instantiate operator would clash and thus make library composition impossible.

These were the original reasons for not having an instantiate operator and I consider them valid against the idea of an instantiation operator that the end user *has to use* and that must be used *exactly once* for each template specialization used. However, would it be possible to have an instantiate operator that could be optionally used? Such an operator would have to have the following properties:

[1] Its use must be optional. That is, if `Vector<int>` is used but isn't explicitly generated, it will be implicitly generated – as required today.

[2] Multiple uses must be ‘‘usually harmless.’’ That is, if two parts of a program instantiates `Vector<int>` they can share one of the resulting instantiations and the other can be ignored or never actually produced.

Note that suppression of multiple copies of an instantiation of a template such as `Vector<int>` is legal because either they are identical in accordance with the one definition rule or else the program is in error for violating the ODR anyway. The tricky issue of enforcement of the ODR is roughly independent of the existence of an explicit instantiate operator.

Note also that some implementations will be able to suppress multiple instantiations (that is be able to link only a single copy into the program), others will not and will therefore have to either issues errors for multiple instantiations or (preferably) suppress instantiation before a linker is faced with multiple copies.

What would be the benefit of an optional instantiation operator?

[1] Users would be able to specify the environment for instantiation.

[2] Users would be able to pre-create libraries of common instantiations in a relatively implementation independent manner.
[3] These pre-created libraries would be independent of changes in the environment of the program that used them (depending only on the context of instantiation).

The link-time and re-compilation efficiency impact could be significant. This would be a semi-manual facility for doing what can be done automatically, but that many implementors have chosen not to do automatically or to provide in different ways so that portability become an issue.

Are there more benefits? Logical problems? Implementation problems?

7 Priorities

Here are my priorities for resolution:

[1] must be solved:
  [1] Name lookup/binding (B).
  [2] Template argument deduction (F1).
  [3] Simple conversions for template arguments (F3).
  [4] Refine specialization rules (S).
  [5] Conversions for function template arguments (F4).

I expect these issues to be resolved along the lines described here given more discussion on the extensions reflector and at the Munich meeting.

[2] Good idea and simple enough to proceed with:
  [1] Explicit specification of template arguments (F2).

I would personally like to see explicit specification of template arguments accepted.

[3] Needs work, maybe more work than we can manage:
  [1] Nested templates (C3).
  [2] Constraints expressed through use (C1).
  [3] Template class overloading based on constraints and number of arguments (C2).
  [4] overloading based on function argument types (F3).

These priorities may be affected by further work on (S) and (I).

8 Acknowledgements

This note is based on half a dozen papers, notably several by Phillipe Gautron, and many dozens of messages on the extensions reflector, including contributions from David Cok, Sean Corfield, Steven Kearns, Patrick Smith, John Max Skaller, and Erwin Unruh. Martin O'Riorden, Andy Koenig, and Jonathan Shopiro have contributed in face-to-face discussions and private email. Peter Juhl and Martin Carroll commented on earlier versions of this note.

9 Appendix A: New Cast Notation

Explicit specification of template arguments in calls provides the syntax suggested for a new notation for type conversion (#92-0122). How well could the semantics suggested for the new casts be simulated by template functions?

```
template<class T, class V> T static_cast(V v) { /* ??? */ }

template<class T, class V> T reinterpret_cast(V v) { /* ??? */; }

template<class T, class V> T const_cast(const V v) { /* ??? */; }
```

Unfortunately, we cannot write the definitions to express the desired semantics exactly. We'd have to use old-style casts and they don't respect access boundaries (they can cast to a private base class) or `const`ness. However,

```
template<class T, class V> T static_cast(V v) { return v; }
```

does most of what is desired. It does all implicit conversions, respect `const`ness and access boundaries, but fail to provide conversions from base to derived. One might consider using constraints and/or overloading to express that. First try, accept pointers only and cast them:

```
template<class T, class V> T static_cast(V* v) { return T(v); }
```

No. This will cast pointers to any type. That is far too powerful. It also would require quite subtle overloading of several `static_cast` template to get anywhere near the desired semantics. Therefore, I consider overloading based on the function argument an unpromising approach.

Second try, use constraints:

```
template<class T, class V>
        constraint {
                V v = 0;
                void* p = v; // V is a non-const pointer type
                *v = *v;     // v is a complete type
                T t = 0;
                const void* pp = t; // T is a pointer type
                *t = *t;     // t is a complete type
                v = t;       // *V is an accessible base of *T
        }
T static_cast(V v) { return T(v); }
```

This `static_cast` will do the unchecked base to derived cast for pointers to complete types only while respecting `const`ness.

Reference casts require the use of the pseudo-function `is_reference`:

```
template<class T, class V>
        constraint {
                is_reference(V);
                is_reference(T);
                V v;
                T t;
                v = t;     // V is an accessible base of T
                           // and V and T are complete types
        }
T static_cast(V v) { return T(v); }
```

Finally, we do ‘‘the rest,’’ that is the standard conversions:

```
template<class T, class V>
        constraint {
                not_reference(V);
                not_pointer(V);
        }
T static_cast(V v) { return v; }
```

I used the pseudo-functions `not_reference` and `not_pointer` to ensure that the constraints were disjoint so that the three `static_cast` templates could be used with a simple overloading scheme.

I'll leave `reinterpret_cast` and `const_cast` as an exercise. I observe that this seems an awfully large amount of mechanism for expressing these fundamentally simple cast operators.