The Problem with Default Arguments for Templates

Bjarne Stroustrup

AT&T

ABSTRACT

The semantics of default arguments of template functions doesn't allow the standard library to work as intented. This note explains the problem and points to a solution.

From a programmer's perspective, I rate this problem the most serious in the current document (of the problems in C++ language and the standard library that we have any hope of addressing now).

1 The Technical Problem

Consider the following example:

```
class<template T> Container {
public:
    Container(size_t n, T def_val = T());  // n elements with value def_val
    // ...
};

Container<string> cs(10);          // default value: string()
Container<string> cs2(10,"411");   // default value: string("411")
```

The `Container` has a constructor that allocates `n` elements and initializes these elements to a default value. By default, that value is the element type's default value. If that value is not suitable, the user can supply a different one.

Now consider a more interesting case:

```
class Point {
    int x, y;
public:
    Point(int xx, int yy) : x(xx), y(yy) { }
    // ... no default constructor ...
}

Container<Point> cp(10);                // error: no default Point()
Container<Point> cp2(19,Point(0,0));  // error: no default Point()
```

Unfortunately, the WP does not allow this last example. The problem is that a default argument is type checked when its declaration is seen. Thus, when the declaration for `Container<Point>` is generated, the declaration

```
Container<Point>::Container(size_t n, Point def_val = Point());
```

is generated, checked, and fails because `Point()` is an error.

This is a serious problem. Not accepting the definition of `cp2` makes `Container` unusable for objects of a type without default constructors, and the designer of `Container` should not impose unreasonable constraints on element types. Indeed, the writer of a library is rarely in the position to impose any but the most minimal constraints on element types developed elsewhere. For many types (for example, `Point` above, iostreams, and handles), default arguments make little sense. Sometimes, an arbitrary

default argument can be a source of errors because it removes the compiler's opportunity to catch a missing or unsuitable initializer. If anything, classes without default constructors ought to be more common than they are today.

This problem is not restricted to constructors. For example, the standard library uses default arguments for `assign()`, `insert()`, and `resize()`.

The problem is not restricted to the standard library. Default arguments are the basis for common programming techniques and a common way of specifying interfaces. With the current semantics these techniques does not carry over to templates.

2 The Magnitude of the Problem

A default argument can be seen as collapsing two (overloaded) functions into one. For example:

```
int f(int i);
int f() { return f(0); }
```

can be seen as *almost* the same as:

```
int f(int i = 0);
```

However, it is more than that. By writing:

```
int f(int i = 0);
```

we state that there really is just one function, and we specify the meaning of a call `f()` to be `f(0)`. Had we used two functions, this commonality of semantics would have been disguised and there would have been two functions to maintain. In other words, when using overloading functions, we specify less about the intented semantics than we do using a default argument in the interface. My experience makes me less than confident that the two functions would retain the intended degree of commonality after a few rounds of maintenance.

The difference between two overloaded functions and one function with a default argument can be observed by taking a pointer to function.

Thus, I consider some uses of default argument good programming (sound software engineering, if you prefer) and not having that facility available for templates would be a loss. It would be a loss even for people who don't use templates because it would be hard to recommend a technique for functions that does not extend cleanly to template functions.

There is about 51 distinct uses of default arguments of template functions in the standard library. About 8 of those are for functions that are not constructors. Beman Dawes has the exact count. As written, the specification of the standard C++ library is not written in C++. This must be resolved somehow.

Fixing the library (however done) solves the problem for a single (but important) case only. Given the popularity of default arguments, the problem will recur daily for years to come.

3 The Solution

The obvious solution is to check the type of a default argument to a template function only if it is used. This is the requirement for template member function bodies (as specified in even the first proposal for templates in 1988).

This is also the requirement for the return type of `operator->()`. Originally, as for default arguments, we adopted the rules for classes for class templates unchanged. Experience then showed that early checking made it unnecessarily hard to write generic classes. For example, providing `->` for smart pointer classes would not be possible without this relaxation of the rules for `operator->()`:

```
template<class T> class Itor {
    T* p;
public:
    T* operator->() { return p; }
    // ...
};

struct S { int m; };

void f(Itor<int> pi, Itor<S> ps)
{
    ps->m = 7;  // fine
    *pi = 2;    // fine, note: pi-> isn't used
}
```

Under the rules for classes, the return type of `operator->()` has to be a something to which `->` can be applied. We relaxed this rule for templates that so that the return type is checked only if `operator->()` is actually used.

Generally, for both C and C++, we try to give errors for actual errors only (rather than for potential ones). A C example is that we are not required to define an declared function that hasn't been called or had its address taken.

As everyone who attended the Nashua meeting and anyone who has dealt with templates semantics knows, even this minor change is non-trivial. There aren't any trivial changes at this stage of the proceedings. It is, however, still minor compared to many other things we have done, and I have no doubt that we can do it well if we set our minds to it. At least one widely-used C++ compiler already implements the intented semantics (or something close to that) rather than the semantics specified in the WP.

4 The Workarounds

In the absence of a solution to the general problem, the standard library must be fixed. I do not consider it acceptable to have the C++ standard library specified using declarations that are not C++. In other words, a statement to the effect that ''use of a default argument in the standard library specification really means that two functions are declared'' is not a suitable technique for specifying the library.

4.1 Adding Overloads

To eliminate default arguments for template functions in the standard library, we have to add about 60 functions. Each function needs a declaration and a specification of its semantics. This implies a minimum of 120 distinct changes scattered over the container, string, and numerics clauses. The minimum number of lines will be on the order of 180 plus blank lines (one declaration plus two lines of semantics per default argument). In a sense we are lucky; had we used default arguments systematically (as I guess we ought to have done), the algorithms clause would have been somewhat shorter and essentially every algorithm would have been affected.

These changes are systematic (just replace every default argument with a function overload), but they are not pretty. For example, without default arguments, the constructors for `map` takes a whole page to declare that way (looking at the SGI implementation).

I do not consider this rewrite an acceptable solution, so I will not dig deep into obscure technical problems. However, please note that adding function overloads does change the set of well-formed programs and silently changes the meaning of some programs that would be well-formed either way:

[1] Adding overloads change the way you can/cannot take pointers to functions.

[2] Moving a call into the body of an overload changes the point of instantiation beyond even the first use of what used to be a default argument. This opens the possibility for different name bindings.

4.2 Rewriting the Default:

It was considered to rewrite default arguments like this:

```
class <template T> T def_fct() { return T(); }

class<template T> Container {
public:
    Container(size_t n, T def_val = def_fct<T>());
    // ...
};
```

This apparently solves the problem by postponing the checking of `T()` until `def_fct<T>()` is used. After all, template class member functions are not analyzed unless used − exactly to avoid requiring every template argument to provide the same set of operations. Surprisingly, this is not supported by the current WP text:

```
Container<Point> cp;                // error: no default T()
Container<Point> cp2(Point(0,0));   // error: no default T(),
                                    // a declaration of a default
                                    // argument is considered a use
```

We ought to fix the definition of *use* so that a call of a function in a default argument of a template function isn't considered a use unless that default argument is used. However, doing so in a general fashion will make the curious `def_fct()` unnecessary by allowing the original `T()`.

Note that this workaround solves the problem only for default arguments that are default values of a template parameter type. Note every occurrence of the problem in the standard library is of this form. In general, many default arguments are not of this form. Requiring circumlocution would add yet another arcane technique to the list of required knowledge of C++ programmers and another burden on teachers.

5 Could we Ignore the Problem?

So what if the standard containers doesn't work with types without default constructors? Then people would have to:

[1] add default constructors (where possible) and thereby limit checking of proper initialization, or
[2] use containers of pointers and thereby increase the amount of free store use, or
[3] avoid the standard containers in favor of some non-standard alternative.

My guess is that

[1] people will add default arguments and their users will get burned, and
[2] this will lead to a burst of pundits explaining the pitfall and deem default arguments fundamentally flawed, and
[3] people will try to add default constructors to every class to make them acceptable as template argument types.

The net effect will be yet another useless debate over an individual language feature that will divert people's attention from major programming issues. The ''myths'' that will inevitable arise from this will confuse people for years and make them nervous. Yet, this will not stop people writing interfaces using default arguments and thereby getting caught by template argument types without default constructors the way the designers of the standard library were.

I don't consider this acceptable, and I note that this will happen even if we ''fix'' the standard library but leave the language unchanged.

6 How Did We Get Into This Mess?

The library was designed based on the assumption that a default argument of a template was checked only if it was used. I was there, and examples like the `Container<Point>` one were presented and assumed to work.

What happened was that we did not review the general language rules systematically enough when templates were added. When designing templates, I spotted the general problem − the rule delaying checking the definitions of used function only reflects that. However, the problem with the return type of operator `->`, and the problem with default arguments took experience to spot.

Once discovered, no action was taken because:

[1] Some people considered the problem unimportant. The extreme version of this argument is ''I don't

like default arguments anyway,'' but several people simply didn't connect the rule about default arguments with the standard library and general programming techniques.

[2] Some people spotted the problem and its relationship to the standard library and thought it obvious that the necessary language change would be made to make the library work as specified.

[3] Some people saw the problem but considered it obvious that the standard library should be changed to fit the language (rather than the other way around).

Basically, everybody thought it was a minor problem that somebody else would easily fix until it is now almost too late to do anything. My impression is that few people (if any) appreciated the whole problems and therefore underestimated it.

The problem was only recently discovered when the WP was clarified so that the wording in section on function arguments was for the first time explicitly related to templates. My guess is that had we spotted the general problem at the time of ''the clarification,'' the clarification would have been to require checking only if used.

7 The Political Problem

From a technical standpoint, a change to the language along the lines presented in §2 is the obviously correct approach to dealing with the default argument problem. Checking default arguments to templates only if used has no negative effects for users. However, this is very late in the standards process.

A change to the standard library will be larger than a change to the language in terms of number of WP lines added and the number of different places touched. However, it ought to be a logically simple change. On the other hand, fixing the standard library simply leaves the problem for users to face in the future. The standard library is an important part of the C++ world but it is only a minute part of the total set of interfaces people design and use.

CD1 and all WPs after that specified the library using default arguments. That represents a promise to users. Secondary documentation based on existing versions of the WP exists and will continue to exist for years to come. This argues against a change to the library because such a change would be much more widespread and prominent than a change to the language making the library definitions valid as written.

8 The Suggested Resolution

Express the rule ''a default argument to a template function is checked only if used'' in proper standardese in the document we hope to vote out as CD2 in London.

Martin O'Riordan (Ireland) and Erwin Unruh (Germany) did a first cut of this in Nashua (X97-0024R1,N1062R1), and Martin volunteered to work on this further before London.