## Why Not Specialize Function Templates?

*This article appeared in****C/C++ Users Journal, 19(7), July 2001****.*

While the title of this article is a question, it could also be made into a statement: this article is about when and why not to specialize templates.

#### The Important Difference: Overloading vs. Specialization

It's important to make sure we have the terms down pat, so here's a quick review.

In C++, there are *class templates* and *function templates*. These two kinds of templates don't work in exactly the same ways, and the most obvious difference is in overloading: Plain old C++ classes don't overload, so class templates don't overload either. On the other hand, plain old C++ functions having the same name do overload, and so function templates are allowed to overload too. This is pretty natural. What we have so far is summarized in Example 1:

// Example 1: Class vs. function template, and overloading   
//

// A class template  
template<class T> class X { /\*...\*/ };      // (a)

// A function template with two overloads  
template<class T> void f( T );              // (b)  
template<class T> void f( int, T, double ); // (c)

These unspecialized templates are also called the underlying *base templates*.

Further, base templates can be *specialized*. This is where class templates and function templates diverge further, in ways that will become important later in this article. A class template can be *partially specialized* and/or *fully specialized*.[**[1]**](http://www.gotw.ca/publications/mill17.htm#1) A function template can only be fully specialized, but because function templates can overload we can get nearly the same effect via overloading that we could have got via partial specialization. The following code illustrates these differences:

// Example 1, continued: Specializing templates   
//

// A partial specialization of (a) for pointer types   
template<class T> class X<T\*> { /\*...\*/ };

// A full specialization of (a) for int   
template<> class X<int> { /\*...\*/ };

// A separate base template that overloads (b) and (c)   
// -- NOT a partial specialization of (b), because   
// there's no such thing as a partial specialization   
// of a function template!   
template<class T> void f( T\* );             // (d)

// A full specialization of (b) for int   
template<> void f<int>( int );              // (e)

// A plain old function that happens to overload with   
// (b), (c), and (d) -- but not (e), which we'll   
// discuss in a moment   
void f( double );                           // (f)

Finally, let's focus on function templates only and consider the overloading rules to see which ones get called in different situations. The rules are pretty simple, at least at a high level, and can be expressed as a classic two-class system:

Nontemplate functions are first-class citizens. A plain old nontemplate function that matches the parameter types as well as any function template will be selected over an otherwise-just-as-good function template.

If there are no first-class citizens to choose from that are at least as good, then function base templates as the second-class citizens get consulted next. Which function base template gets selected depends on which matches best and is the "most specialized" (important note: this use of "specialized" oddly enough has nothing to do with template specializations; it's just an unfortunate colloquialism) according to a set of fairly arcane rules:

If it's clear that there's one "most specialized" function base template, that one gets used. If that base template happens to be specialized for the types being used, the specialization will get used, otherwise the base template instantiated with the correct types will be used.

Else if there's a tie for the "most specialized" function base template, the call is ambiguous because the compiler can't decide which is a better match. The programmer will have to do something to qualify the call and say which one is wanted.

Else if there's no function base template that can be made to match, the call is bad and the programmer will have to fix the code.

Putting these rules together, here's a sample of what we get:

// Example 1, continued: Overload resolution   
//   
bool b;   
int i;   
double d;

f( b );        // calls (b) with T = bool   
f( i, 42, d ); // calls (c) with T = int   
f( &i );       // calls (d) with T = int   
f( i );        // calls (e)   
f( d );        // calls (f)

So far I've deliberately chosen simpler cases, because here's where we step off into the deep end of the pool.

#### Why Not Specialize: The Dimov/Abrahams Example

Consider the following code:

// Example 2: Explicit specialization   
//   
template<class T> // (a) a base template   
void f( T );

template<class T> // (b) a second base template, overloads (a)   
void f( T\* );     //     (function templates can't be partially   
                  //     specialized; they overload instead)

template<>        // (c) explicit specialization of (b)   
void f<>(int\*);

// ...

int \*p;   
f( p );           // calls (c)

The result for the last line in Example 2 is just what you'd expect. The question of the day, however, is *why* you expected it. If you expected it for the wrong reason, you will be very surprised by what comes next. After all, "So what," someone might say, "I wrote a specialization for a pointer to int, so obviously that's what should be called" - and that's exactly the wrong reason.

Consider now the following code, put in this form by Peter Dimov and Dave Abrahams:

// Example 3: The Dimov/Abrahams Example   
//   
template<class T> // (a) same old base template as before   
void f( T );

template<>        // (c) explicit specialization, this time of (a)  
void f<>(int\*);

template<class T> // (b) a second base template, overloads (a)   
void f( T\* );

// ...

int \*p;   
f( p );           // calls (b)! overload resolution ignores   
                  // specializations and operates on the base   
                  // function templates only

If this surprises you, you're not alone; it has surprised a lot of experts in its time. The key to understanding this is simple, and here it is: *Specializations don't overload.*

Only the base templates overload (along with nontemplate functions, of course). Consider again the salient part from the summary I gave above of the overload resolution rules, this time with specific words highlighted:

…

If there are no first-class citizens to choose from that are at least as good, then *function base templates* as the second-class citizens get consulted next. Which *function base template* gets selected depends on which matches best and is the "most specialized" […] according to a set of fairly arcane rules:

If it's clear that there's one "most specialized" *function base template*, that one gets used. If that *base template* happens to be specialized for the types being used, the specialization will get used, otherwise the base template instantiated with the correct types will be used.

... etc.

Overload resolution only selects a base template (or a nontemplate function, if one is available). Only after it's been decided which base template is going to be selected, and that choice is locked in, will the compiler look around to see if there happens to be a suitable specialization of that template available, and if so that specialization will get used.

#### Important Morals

If you're like me, the first time you see this you'll ask the question: "Hmm. But it seems to me that I went and specifically wrote a specialization for the case when the parameter is an int\*, and it is an int\* which is an exact match, so shouldn't my specialization always get used?" That, alas, is a mistake: If you want to be sure it will always be used in the case of exact match, that's what a plain old function is for -- so just make it a function instead of a specialization.

The rationale for why specializations don't participate in overloading is simple, once explained, because the surprise factor is exactly the reverse: The standards committee felt *it would be surprising that, just because you happened to write a specialization for a particular template, that it would in any way change which template gets used*. Under that rationale, and since we already have a way of making sure our version gets used if that's what we want (we just make it a function, not a specialization), we can understand more clearly why specializations don't affect which template gets selected.

**Moral #1:** If you want to customize a function base template and want that customization to participate in overload resolution (or, to always be used in the case of exact match), make it a plain old function, not a specialization. And, if you do provide overloads, avoid also providing specializations.

But what if you're the one who's writing, not just using, a function template? Can you do better and avoid this (and other) problem(s) up front, for yourself and for your users? Indeed you can:

**Moral #2:** If you're writing a function base template, prefer to write it as a single function template that should never be specialized or overloaded, and then implement the function template entirely as a simple handoff to a class template containing a static function with the same signature. Everyone can specialize *that* -- both fully *and partially*, and without affecting the results of overload resolution.

// Example 4: Illustrating Moral #2   
//   
template<class T>   
struct FImpl;

template<class T>   
void f( T t ) { FImpl<T>::f( t ); } // users, don't touch this!

template<class T>   
struct FImpl   
{   
  static void f( T t ); // users, go ahead and specialize this   
};

#### Summary

It's okay to overload function templates. Overload resolution considers all base templates equally and so it works as you would naturally expect from your experience with normal C++ function overloading: Whatever templates are visible are considered for overload resolution, and the compiler simply picks the best match.

It's a lot less intuitive to specialize function templates. For one thing, you can't partially specialize them -- pretty much just because the language says you can't.[**[2]**](http://www.gotw.ca/publications/mill17.htm#2) For another thing, function template *specializations don't overload*. This means that any specializations you write will not affect which template gets used, which runs counter to what most people would intuitively expect. After all, if you had written a nontemplate function with the identical signature instead of a function template specialization, the nontemplate function would always be selected because it's always considered to be a better match than a template.

If you're writing a function template, prefer to write it as a single function template that should never be specialized or overloaded, and implement the function template entirely in terms of a class template. This is the proverbial level of indirection that steers you well clear of the limitations and dark corners of function templates. This way, programmers using your template will be able to partially specialize and explicitly specialize the class template to their heart's content without affecting the expected operation of the function template. This avoids both the limitation that function templates can't be partially specialized, and the sometimes surprising effect that function template specializations don't overload. Problem solved.

If you're using someone else's plain old function template (one that's not implemented in terms of a class template as suggested above), and you want to write your own special-case version that should participate in overloading, don't make it a specialization; just make it an overloaded function with the same signature.

#### Acknowledgments

Thanks to Peter Dimov and Dave Abrahams for prompting me to write about this topic and offering the key example, and to John Spicer for helping me get my head around the rationale for why things are the way they are.

#### Notes

1. In standardese, a full specialization is called an "explicit specialization."

2. There is some discussion going on within the committee about potentially allowing function template partial specialization in the next version of the C++ standard, whose work is just getting under way.