Dr. Dobbs Article: C++ Templates

By Pete Becker, August 01, 1993

Post a Comment

Source Code Accompanies This Article. Download It Now.

- cpptmp.asc
- varray.asc
- varray.zip

One of the benefits of C++ templates is they make it easier for you to write custom tools. From parameter types to instantiation, Pete turns templates inside-out, while Doug Reilly builds a C++ virtual-array template class to show how templates can be used.

SIMPLIFY YOUR C++ CODE WITH TEMPLATES

Pete is a software engineer at Borland International and is Borland's principal representative to the ANSI C++ Committee. At Borland he works on Object Windows Library, class libraries, and sometimes linkers. He can be contacted at 1800 Green Hills Road, Scotts Valley, CA 95066.

Where would the automobile industry be without custom tools? An assembly line in Detroit turns out hundreds of cars each day, each having its own particular variations of color, engine size, and accessories, all built according to a common plan. Since they use the same plan every time, automobile engineers can create custom tools specifically for that design, simplifying the manufacturing of cars and multiplying the productivity of factories.

C++ templates can multiply your coding productivity by making it easier to write your own custom tools. You're not producing hundreds of classes each day, so your productivity gains probably won't be as dramatic as those in the automobile industry. Still, a few well-designed templates in your toolkit can make your job much easier.

Life Without Templates

Suppose you've written a class that implements a stack of integers like Example 1. Of course, programming being what it is, the next time you have to write a stack, it won't be a stack of integers but, say, a stack of strings. You could reuse *IntStack* by copying it, changing its name, and replacing *int* with a string where appropriate. However, this approach has drawbacks.

First, it's error prone. You can't just do a global search and replace. If you do, you'll end up with the member function *ItemsInStack* returning a string, which probably isn't what you want. Doing this transformation correctly requires you to understand every use of *int* in the original class. Transforming a larger, seldom-used class could prove to be an enormous undertaking. Not to mention that once you've done this a few times, there'll be several variations of stacks used by different programs on your hard disk. If you find and fix a logic error in one, you ought to fix it in the others as well. Unless you've kept scrupulous notes on where these variations are used, you'll have a hard time finding them all.

Some programmers reject templates, saying a stack class isn't very big and you can write one when needed. Sure you can copy the definition of *IntStack* and, with some care, change it to a *FloatStack* in minutes--and there's a good chance it will work correctly. However, it takes less time to create a *FloatStack* from a template: You *#include "stack.h"* and *typedef Stack<float> FloatStack*. These two lines of code do the same as the 28 lines in Example 1. Don't let anyone tell you that the other way is better.

On the Inside

These two lines of code don't tell the whole story. Consider the STACK.H header in Example 2, the heart of the *Stack* template. If you compare this template with the *IntStack* class, you'll see the two are almost identical. The template begins with *template <class Type>*, which tells you it's a template that takes one parameter we'll refer to as *Type*. Most of the *int*s in the original class have been replaced with *Type*s in the template, the exception being the *int* specifying the return type of *ItemsInStack*. Does that sound familiar? It's the same transformation that changes an *IntStack* into a *FloatStack*. In fact, it's common to develop a template by first writing a class that does what's needed for a single data type, then changing that class into a template by replacing that data type with a parameterized type as we did here. When you create a template, you only have to do the transformation once. After that, the compiler can do it for you.

A Brief Diversion For Philologists

The chapter on templates in *The Annotated C++ Reference Manual* (*ARM*) includes some rather confusing terms, such as "class templates," "template classes," "function templates," and "template functions." My *Stack* example is a class template--a template used to create classes. I used it to create a *Stack*

<float>, a class created from a template--a template class.

The same applies to function templates, which, as the name suggests, are templates that create functions. The template that you write is a function template. When you use that template to create a function, that function is a template function.

Template Parameters

Every template definition begins with the keyword template followed by a template parameter list. As in the *Stack* example, the template parameter is delimited by a less-than (<) and a greater-than (>) symbol. The alternative of overloading the meanings of {}, [], or () results in confusing code because these delimiters already have so many other meanings.

The parameters in a template definition fall into two broad categories: type parameters, which tell the compiler to expect the name of a type when the template is used; and nontype parameters, which tell the compiler to expect a value. For instance, Example 3 changes the *Stack* template, so you could specify how large the stack should be. The difference between this and Example 2 is that I've added a parameter named *Size* to the template definition and removed the *enum* that defined *Size* in the earlier version.

In Example 3, the first parameter is a type parameter, the second, a nontype parameter. When the class keyword is used in a template parameter list, it indicates that the name following it is a type parameter. This isn't the same as the use of class in a class definition. It doesn't mean that when the template is used, the argument must be the name of a class; it only means that it must be the name of a type.

When using a template, you put an argument list after the name of the template. That argument list contains the actual names or values that the compiler should use when it expands the template. Where the parameter list contains type parameters, the argument list must contain type names. Where the parameter list contains nontype parameters, the argument list must contain values of the type specified for the parameter. You'd use the two-parameter version of the *Stack* by specifying the type of the values and the maximum number of entries that the *Stack*holds: *Stack<int,20>* is a stack of integers with a capacity of 20, just like the original definition.

Class Templates Create Types

When you use a class template, you're creating a type name. Don't get confused because it's a template--it acts just like any other type name. All the statements in Example 4 are legal, for example, and they mean just what you'd expect. When using templates, keep in mind that they may seem new and strange and have some odd-looking syntax, but they usually fit into your code

without requiring you to do anything differently. That's the beauty of good tools: Once you understand how to use them, they get out of your way and let you do your job.

Gotchas

Templates have a couple of syntactic quirks you should watch out for. The last line in the list of declarations in Example 4 creates a stack of stacks. It would be an error to declare *Stack<Stack<int>> StackOfStacks*. The problem is that the two greater-thans (>>) after *int* in this declaration don't terminate the two template-argument lists. C++ uses C's "maximum-munch" rule, so the >> is interpreted as a shift-right operator. The compiler will give you strange messages if you omit the space between > and >.

Parameter-list delimiters can also be a problem if you use arithmetic expressions for nontype parameters. Using the two-parameter version of the *Stack* class, you might try to write Example 5(a). The problem here is that the first > is the end of the argument list, so everything after that is nonsense. The way around this is to put the size expression in parentheses; see Example 5(b).

Finally, you're not allowed to overload a class-template name. Consequently, you couldn't use the *Stack<class Type>* and *Stack<class Type,int Size>* templates in one program.

Function Templates

Function templates can be used to create functions. The rules for dealing with function templates are more complicated than those for class templates because functions and function templates can be overloaded and because we don't use an explicit template-parameter list when calling a template function. Consider Example 6, the compiler has to figure out what the template parameters are from the arguments used in the function call. In *Show(f)*, the argument *f* is of type *float*. Looking back at the template definition, you see that the only parameter used in the function call is the template parameter *Type*. The compiler infers that *Type* must correspond to *float* and acts accordingly. *Show(f)* acts just as if you'd written it explicitly to handle floats; see Example 6(b).

Parameter Types

One problem you can run into with function templates is the rule that the compiler does not do conversions on the arguments to a function template. This is a case where your intuition may lead you astray. The template in Example 7(a) calculates the lesser of two values. Example 7(b)uses this template for some calculations. Here, *min()* is called with two arguments of type *int*. The compiler looks at the template, sees that it takes two parameters of the same type, and figures out that if it replaces *T* with *int*, it has a match. The compiler knows there's a function *int min(int,int)* and calls it. Example 7(c) is similar to 7(b): The compiler knows from the template that there's a function *long min(long,long)* and calls it.

Example 7(d), however, isn't like the previous examples. The two types in the call to *min()* are different. The function template takes two parameters of the same type, so the compiler cannot use the template and the call is an error. The compiler doesn't know of any function named *min()*that takes a *long* and an *int*. If I'd simply written the usual *#define min(a,b) ((a)<(b)?(a):(b))*, this call would have worked. The compiler would see that it was comparing an *int* and a *long*, and would perform the usual arithmetic conversions, converting the *int* argument to a *long* so that the two matched, and the result would be a *long*. That doesn't happen with templates because of the "no-conversion" rule.

It's generally agreed that the no-conversion rule is too harsh, and will likely change when the ISO C++ committee reviews the rules on templates. Indeed, many of today's compilers, including cfront, allow some conversions even though they're technically illegal; don't be surprised if some compilers complain about them.

Overloading

Still, it's possible to use the example *min()* function template to compare an *int* and a *long*. To understand how, look at how the compiler handles overloaded functions. Consider the best-match

rules in section 13.2 of the *ARM*. These rules define the algorithm the compiler is supposed to use to determine which function to call when several functions have the same name. This rule was written before templates were added to the language. Templates don't change the rule, they extend it. The new rule states:

A template function may be overloaded either by (other) functions of its name or by (other) template functions of that same name. Overloading resolution for template functions and other functions of the same name is done in three steps:

[1] Look for an exact match (sec. 13.2) on functions; if found, call it.

[2] Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.

[3] Try ordinary overloading resolution (sec. 13.2) for the functions; if a function is found, call it.

According to the first sentence, the declarations in Example 8(a) can all be used in the same program. You end up with two different templates with the same name, *min()*, and an ordinary function prototype for a function with this name. That they may conflict isn't important at this point. Only when you try to use one of these functions must the compiler worry about ambiguities. The rest of the rule addresses this point. For example, suppose the program contains two string variables, *s1* instantiated to *"abcd"* and *s2* instantiated to *"efgh"*, and calls the function *min(s1, s2)*. The call to *min()* supplies two arguments, both of type string. Declaration #2 in Example 8(a) says there's a function, *min()*, that takes two arguments of type *string*. So, you have a function that exactly matches the call. According to Step #1 of the overloading rule, you don't have to go any further.

Next, consider two integer variables, *i1=3* and *i2=4*, and a call to *min(i1, i2)*. Because the two arguments are of type *int*, Step #1 of the overloading rule doesn't help here--we don't have any function that exactly matches the call. Moving to Step #2, there's "a function template from which a function that can be called with an exact match can be generated," namely, the template *min*

(T t1, T t2). The compiler can use that template to generate *min(int,int)*. That function exactly matches the call, so that's the function that will be called.

Now consider a variable string *s1="abcd"* and a variable *char *s2="defg"*. The call string *min(s1, s2)* doesn't satisfy either Step #1 or #2. We haven't seen the definition of this string class, but from its use you've probably figured that it has a constructor that takes an argument of type *char**. The constructor is used to create the auto strings we've been passing as parameters. The compiler could use that constructor to call *min(string,string)* by creating a temporary object of type *string* and using that temporary object as the second parameter. That's just an ordinary function call, and it's done just as if the template weren't present. Step #3 says that's what the compiler should do in this case.

Step #3 also tells us what we need to do to persuade the compiler that it's okay to call *min()* with an *int* and a *long*--tell the compiler there's a version of *min()* that takes two *long*s. Once that's done, the compiler will use Step #3 of the overloading rule and promote the *int* parameter to a *long*. You don't have to supply a definition of this particular variation as long as the compiler can generate it from the template. By supplying a prototype, you make the call legal. By supplying the template definition, you give the compiler a way of producing the actual function.

Template Specializations

Step #2 of the overloading rule says to, "look for a function template from which a function that can be called with an exact match can be generated; if found, call it." It doesn't say "generate the function and call it." That's quite deliberate. The compiler simply generates a call to the function that has that name. You can provide your own version of that function somewhere else in the code; if you do, that function will be called when the program is run.

Consider Example 8(b), a variation of the *min()* example, which, though admittedly contrived, is supposed to take two command-line parameters (text representations of numbers) and return the lower value of the two. Unfortunately, it doesn't work because the *min()* template, when applied to two *char**s, compares the pointers, not the C strings they point to. This is where specialization comes to the rescue: You can provide your own definition of *min(char *, char *)* that calls *strcmp()* instead of comparing pointers; see Example 8(c).

If this function appears anywhere in the program, even in a different module from the one that contains the call, then it should be invoked at run time. The template should not be used. Think of template instantiation as a desperate measure, to be used only when nothing else works. The compiler uses the overloading rules to figure out what call to make, then tries to find an ordinary function to handle the call. If there's no ordinary function, it will instantiate the template.

Forward References

Notice in Example 8 that I used a forward reference to the *min()* template. This works much like a function prototype, telling the compiler there's a function template named *min()* that takes two parameters of the same type and returns a result of that type. That's all the compiler needs to know to call functions created with this template. Of course, it eventually has to see the actual definition, but the definition doesn't have to appear until later; see Example 9. When you don't want to or can't expose the template definition, you can tell the compiler that some name is, in fact, the name of a template. Of course, if you haven't given the compiler the definition of the *Stack* template, you can't use that template in any way that depends on its definition.

Template Instantiation

But what happens when the compiler actually expands the template? Where does it put the code that the template creates? These details depend on the compiler.

cfront, for example, requires you to put class-template declarations and corresponding inline functions in a header file, and non-inline member-function definitions in a file with the same base name as the header file, located in a special directory so that the compiler can find it. Each time the compiler instantiates a template, it sticks the code into a special area known as a "repository." During linking, the linker looks in the repository for the template instantiations that it needs. If a particular instantiation isn't there, the linker has to call the compiler to create it.

Borland C++, on the other hand, requires you to put everything in the header file. It compiles the code directly into the .OBJ file where it's used. The linker combines duplicate template instantiations, so your executable file only ends up with a single copy of the code for any particular instantiation. Whatever compiler you use, once you've set things up properly, you shouldn't have to do anything special to get templates instantiated. The compiler should take care of it.

Templates in Your Tool Kit

One of the greatest advantages of C++ over C is that it supports inheritance, which allows you to factor out common features into single class. This improves maintainability and gives you a set of reusable building blocks for future use.

Templates extend this form of generalization by enabling the creation of families of similar classes and functions. Your understanding of a problem often changes during the development of an application, which means you constantly adapt building blocks to the changing requirements. Having a well thought-out set of templates makes these adaptations much easier and much less error prone.

References

Ellis, Margaret and Bjarne Stroustrup. *The Annotated C++ Reference Manual*, second edition. Reading, MA: Addison-Wesley, 1992.

```
1
```

Example 1: A C++ class that implements an integer stack.

```
class IntStack
{
public:
    IntStack() : Current(Data) {}
    int ItemsInStack() const
        {
        return Current - Data;
        }
    int Top() const
        {
        assert( ItemsInStack() != 0 );
        return *Current;
        }
    void Push( int i )
        {
        assert( ItemsInStack() < Size );
        *Current++ = i;
        }
    int Pop()
        {
        assert( ItemsInStack() != 0 );
        return *(++Current);
        }
private:
    enum { Size = 20; };
    int Data[Size];
    int *Current;
};
```

Example 2: Stack.h contains the Stack template.

```
template <class Type> class Stack
{
public:
    Stack() : Current(Data) {}
    int ItemsInStack() const
        {
        return Current - Data;
        }
    Type Top() const
        {
        assert( ItemsInStack() != 0 );
        return *Current;
        }
    void Push( Type t )
        {
        assert( ItemsInStack() < Size );
        *Current++ = t;
        }
    Type Pop()
        {
        assert( ItemsInStack() != 0 );
        return *(++Current);
        }
private:
    enum { Size = 20; };
    Type Data[Size];
    Type *Current;
};
```

Example 3: Changing the Stack template so that its size can be specified.

```
template <class Type, int Size> class
Stack
{
public:
    // same as above
private:
    Type Data[Size];
    Type *Current;
};
```

Example 4: Legal declarations using the Stack template.

```
#include <stack.h>

Stack<int> S1, S2;
Stack<int> *Ptr;
extern Stack<int> S3;
void f( Stack<int>& );
class DerivedStack : public Stack<int> {};
Stack< Stack<int> > StackOfStacks;
```

Example 5: (a) Attempting to use arithmetic expressions in the two-parameter version of the Stack class; (b) placing the size expression within parentheses avoids the parameter-list delimeter problem in 5(a).

```
(a)

const Count = 17;
Stack< int, Count>10?Count:10 > Data;


(b)

Stack< int, (Count>10?Count:10) > Data;
```

Example 6: (a) Usage for function templates; (b) writing this function once with a template is much less time consuming than writing it for every data type.

```
a)

template <class Type> void Show( Type T )
{
    Type times2 = T*2;
    cout << T << '\t' << times2 << endl;
}

int main()
{
    float f = 3.14159;
    double d = 2.1;
    int i = 3;
    Show(f);
    Show(d);
    Show(i);
    return 0;
}

(b)

void Show( float f )
{
    float times2 = f*2;
    cout << f << '\t' << times2 << endl;
}
```

Example 7: (a) Function template to calculate the lesser of two values; (b) the compiler looks at the template in 7(a), replaces T with int, and calls int min(int,int); (c) in this case the compiler calls long min(long,long); (d) because the two types in the call to min() are different, the compiler cannot use the template and the call is an error.

```
(a)

template <class T> T min( T t1, T t2 )
{
    return t1<t2 ? t1 : t2;
}


(b)

int GetMin( int i1, int i2 )
{
    return min( i1, i2 );
}


(c)

long GetMin( long l1, long l2 )
{
    return min( l1, l2 );
}


(d)

long GetMin( long l1, int i1 )
{
    return min( l1, i1 );
}
```

Example 8: (a) These declarations can all be used in the same program; (b) a variation on the template class in 8(a); (c) definition of min(char *, char *) that calls strcmp() instead of comparing pointers.

```
(a)

template <class T>
T min( T t1, T t2 );            // 1
string min( string,string );    // 2
template <class T>
T min( T t1, T t2, T t3 );     // 3


(b)

template <class T>
T min( T t1, T t2 )
{ return t1 < t2 ? t1 : t2; }

int main( int argc, char *argv[] )
{
    return atoi( min(argv[1],argv[2]) );
}

(c)

char *min( char *s1, char *s2 )
{
    if( strcmp( s1, s2 ) < 0 )
        return s1;
    else
        return s2;
}
```

A Virtual-array Class using C++ Templates

Douglas Reilly

Doug owns Access Microsystems, a software-development house specializing in C/C++ software development. He is also the author of the BTFILER and BTVIEWER Btrieve file utilities. Doug can be contacted at 404 Midstreams Road, Brick, NJ 08724, or on CompuServe at 74040,607.

Class templates allow the use of a single set of functions to perform identical operations on an unlimited set of types. Templates were first described as "experimental" in *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup (Addison-Wesley, 1990). Templates are currently available in Borland C++ 3.x, and should soon be available in all C++ compilers. It's important to note that templates may not provide any savings in the resulting object code, and in this way might be no more efficient than macros. But machine efficiency isn't the only concern in programming. In the machine/

programmer cost equation, the programmer is more expensive, and eliminating problems associated with macros makes the programmer more productive.

On more than one occasion, I've hit the memory "wall" when adding features or capacity. I'm primarily an MS-DOS programmer, and have long envied programmers in other environments who could declare arrays much larger than would fit in the physical memory of the machine. Templates, combined with creative use of operator overloading, are the answer.

virtArray is a virtual-array template class; see Listings One and Two (page 102). It buffers a user-selectable number of elements, but stores the balance of the array on disk. Because of the ability to overload the subscript operator ([]) and return references, these virtual arrays can replace existing arrays with only minor modifications to the source code. After a while, you can even forget they're not part of the C++ language proper.

To help test performance and determine the degree of difficulty in integrating *virtArray* into an existing application, I turned to an existing MS-DOS application that used an array for storing moderate-sized structures that defined fields in a file. This program was the perfect candidate for a virtual array, since there was a need to use more fields than was possible with traditional arrays.

The first change was the declaration. I used a *typedef* to hide the details (template arguments, and so on) and to ensure that exactly the same template arguments were specified each time the virtual array was declared. This is important since template arguments must match exactly in the definitions and declarations if you are to reference the same object.

Notice the *nullOut()* member function. A common way to initialize an array is to use *memset()*. This won't work with a virtual array. Given a virtual array like *virtArray<int,256>intArray(20000);*if you *memset()&intArray[0]* for 20,000 elements times *(sizeof(int))*, you'll initialize the 256 data elements of the cache, and then initialize the next 39,488 bytes in memory. After rebooting your machine, replace the *memset()* with a call to the *nullOut()* member function. This function initializes the cache as well as the data elements in the temporary disk file.

Another change in behaviors between traditional arrays and the virtual array is that you cannot simply send the address of element 0 to a function and expect that to act as a pointer to the whole array. At best, this will enable the receiving function to get at *cacheSize* elements of the array, and will then access whatever is next in memory, just as in the case of the *memset()*described earlier.

An implementation detail that significantly effects performance is the way the cache is implemented. If an array element not currently in the cache is requested in the *operator []*member function, the call to the *get()* member function gets the requested element and the next *cacheSize--1* element. This is fairly effective in many cases, since a common construct is to loop through an array starting at 0, looking for the end of an array or simply processing each element. If the looping through the array takes place from the end of the array back to 0, this will force a disk read for each call to *operator []*. A reasonable alternative would be to fill the cache so that half the elements cached would be before the current element and half would be after.

Example 9: A forward reference to a template works much like a function prototype.

```
template <class T>
T min( T t1, T t2 );

int test( int i1, int i2 )
{
    return min( i1, i2 );
}
template <class T>
T min( T t1, T t2 )
{
    return t1 < t2 ? t1 : t2;
}
```

[LISTING ONE]

12