This is a pre-publication draft of the article I wrote for the November 2002 issue of *C/C++ Users Journal*. "Pre-publication" means this is what I sent to *CUJ*, but it may not be exactly the same as what appeared in print, because *CUJ* and I typically make small changes during preparation of the final version of the article.

Class Template, Member Template — or Both?

by Scott Meyers

Sometimes what you templatize makes all the difference.

The STL is wonderful, but in places it demands gratuitous redundancy. For example, consider this code:

```
vector†<int>  v;
deque<int> d;

...

transform(
  v.begin(), v.end(),      // do d[i] *= v[i] for all i
  d.begin(),
  d.begin(),
  multiplies<int>());
```

Even though I'm multiplying two ints together, I have to explicitly specify that I want to perform integer multiplication. This is more than annoying, because if I later change v and d to hold a different data type (e.g., double), I must also remember to update all the places where I've used multiplies (or plus, etc.) to specify the new data type. If I don't, I end up with code like this, which almost certainly doesn't do what I want:

```
vector<double> v;
deque<double> d;

...

transform(
  v.begin(), v.end(),      // do d[i] *= v[i] for all i,
  d.begin(),               // but use integer
  d.begin(),               // multiplication!
  multiplies<int>());
```

† Throughout this article, I will ignore the need to specify that standard components are in the std namespace.

The reason I must specify a type for multiplies is that it is a class template:

```
template<typename T>
struct multiplies:
  binary_function<T, T, T> {

  T operator()(
    const T& x,
    const T& y) const
  { return x*y; }

};
```

Unlike class templates, function templates offer implicit type deduction, so the usual approach to eliminating the need to specify types for class templates is to write a helper function template that deduces the type(s) and returns the appropriate type of object as the return value. For multiplies, such a helper function template might look like this:

```
template<typename T>
inline
multiplies<T>
  make_multiplies(
    const T&, const T&)
{ return multiplies<T>(); }
```

In many cases, this works fine, and the general approach is enshrined in the standard in the form of the helper template make_pair (which generates pair objects). The approach has its limitations, however. In the case of make_multiplies, notice how callers must pass two objects of type T, even though neither object is used; only the type is. Furthermore, it's hard to see how to use make_multiplies with the earlier transform call, because the arguments are available only *inside* transform:

```
transform(
  v.begin(), v.end(),
  d.begin(),
  d.begin(),
  make_multiplies(???, ???));
```

When running up against difficulties like this, I have found the following transformation useful: instead of creating a class (or struct) template with non-template member functions, consider creating a non-template class with a member template instead. That is, consider moving the templatization from the class to the class's member function(s).

If we apply this idea to multiplies, we come up with this:

```
struct NewMultiplies {        // no longer a template
```

```
  template<typename T>      // now a template
    T operator()(
      const T& x,
      const T& y) const
    { return x*y; }

};
```

Callers would use it like this:

```
vector<int> v;
deque<int> d;

...

transform(
  v.begin(), v.end(),      // do d[i] *= v[i] for all i
  d.begin(),
  d.begin(),
  NewMultiplies());        // deduce that integer
                           // multiplication is
                           // appropriate
```

Not only is this more robust if the types stored in v and d change, it's also less to type.

This idea — putting a templatized member function (often operator()) inside a non-template class — is the basis of my favorite little trick in *Effective STL* [1], namely, a generic pointer-deleting functor:

```
struct DeleteObject {

  template<typename T>
  void operator()(
    const T* ptr) const
  { delete ptr; }

};

Container<T*> c;

...

for_each(
  c.begin(), c.end(),      // delete pointers in c
  DeleteObject());         // regardless of type
```

Useful as this technique is, sometimes you *want* to specify the type explicitly. For example, suppose v and d hold doubles, but for some reason you want to perform integer multiplication. In that case, you want to say this:

```
vector<double> v;
deque<double> d;

...
```

```
transform(
  v.begin(), v.end(),      // do d[i] *= v[i] for all i,
  d.begin(),               // but use integer
  d.begin(),               // multiplication!
  multiplies<int>());
```

With NewMultiplies, you don't have that option. Furthermore, because NewMultiplies fails to inherit from binary_function, NewMultiplies objects aren't adaptable, and that means that while you can use adapters like bind2nd with multiplies, you can't use them with NewMultiplies:

```
vector<int> v;

...

transform(
  v.begin(), v.end(),
  v.begin(),                          // fine, multiplies v's
  bind2nd(multiplies<int>(), 2));     // elements by 2

transform(
  v.begin(), v.end(),
  v.begin(),                          // error! Can't apply
  bind2nd(NewMultiplies(), 2));       // a binder to a
                                      // NewMultiplies object
```

These limitations make NewMultiples less attractive than it initially seemed.

What we really want, of course, is the best of both worlds:

- Implicit type deduction when it does the right thing.
- Explicit type specification when we need it, e.g., when we deliberately want to use the "wrong" type or when we need to employ a binder.

Fortunately, it is easy to achieve this blissful state. To get implicit type deduction, we use the NewMultiplies approach: a non-template struct with a templatized operator() member function. To allow explicit type specification, we use the multiplies approach: a templatized struct with a non-template operator(). The only twist is that we combine them: we make the non-template NewMultiplies-like struct a total specialization of the multiplies-like struct. It looks like this:

```
template<typename T = void>
struct NewMultiplies:
  std::binary_function<T, T, T> {

  T operator()(
    const T& x,
    const T& y) const
  { return x*y; }
```

```
};

template<>
struct NewMultiplies<void> {
   template<typename T>
   T operator()(
      const T& x,
      const T& y) const
   { return x*y; }

};
```

Notice that the general NewMultiplies template takes a single type argument, just like multiplies does, and NewMultiplies is implemented exactly like multiplies is. Unlike multiplies, however, NewMultiplies' type argument is optional. When that argument is omitted, the type defaults to void. We've explicitly specialized NewMultiplies<void>, however, and that class contains a member function template for operator(). Result? When we pass NewMultiplies a type argument, it behaves exactly like multiplies. When we use NewMultiplies without a type argument, it uses implicit type deduction to figure out the type of multiplication to perform. Here are some examples:

```
vector<double> v;
deque<double> d;
...
transform(
   v.begin(), v.end(),
   d.begin(),
   d.begin(),
   NewMultiplies<>());              // deduce that T is double

transform(
   v.begin(), v.end(),
   d.begin(),
   d.begin(),
   NewMultiplies<int>());           // do integer multiplication

transform(
   v.begin(), v.end(),
   v.begin(),
   bind2nd(
      NewMultiplies<double>(), 2)  // do double multiplication
);
```

As you can see, NewMultiplies gives you everything multiplies gives you, plus the ability to avail yourself of implicit type deduction.

You could write similar templates and template specializations for the other standard multiplies-like functors (e.g., plus, minus, etc.), but there's no point in expending that much energy, because Donovan

Rebbechi has pointed out that you can automate the process[2]. Just create a template, NewFunctor, such that NewFunctor<multiplies, T> behaves just like multiplies<T> (i.e., it offers no templatized operator()) while NewFunctor<multiplies> behaves just like our original NewMultiplies (i.e., it does have a templatized operator()). NewFunctor will also work with plus, minus, etc., which is what makes it worth writing.

Here it is:

```
template < template <typename> class Functor,     // NewFunctor<F,T>
           typename T = void>                     // inherits from F<T>,
struct NewFunctor: Functor<T> {};                 // hence acts like it —
                                                  // except when T=void

template <template <typename> class Functor>      // NewFunctor<F, void>
struct NewFunctor<Functor, void> {                // and NewFunctor<F>
                                                  // act like NewF<>
  template <class T>
  T operator() (const T& left, const T& right) const
  {
    return Functor<T>()(left,right);
  }
};
```

The only part of this that might give you pause is the use of a template template parameter in the definition of NewFunctor:

```
template < template <typename> class Functor,
           typename T = void>
struct NewFunctor: Functor<T> {};
```

This just says that NewFunctor's first parameter must be a template taking one type argument. The name of the type parameter for this template is omitted. An alternative way to write the same thing is

```
template < template <typename U> class Functor,
           typename T = void>
struct NewFunctor: Functor<T> {};
```

but since there's no way to take advantage of the name U anywhere, the name is customarily omitted. (Andrei Alexandrescu has demonstrated that template template are far from a mere syntactic curiosity; they are a powerful part of C++. For the full story on this, consult Alexandrescu's book, *Modern C++ Design* [3].)

Here's NewFunctor in action [4]:

```
vector<double> v;
deque<double> d;
...
```

```
transform(
  v.begin(), v.end(),
  d.begin(),
  d.begin(),
  NewFunctor<multiplies>());                  // deduce that T is double

transform(
  v.begin(), v.end(),
  d.begin(),
  d.begin(),
  NewFunctor<plus, int>());                   // do integer addition

transform(
  v.begin(), v.end(),
  v.begin(),
  bind2nd(
    NewFunctor<minus, double>(), 2)           // do double subtraction
  );
```

It is not difficult to generate similar wrappers for the relational operators (e.g., less, greater, equal_to, etc.). With some additional effort, it is possible to give NewFunctor support for mixed-type operations (e.g., multiplying ints and doubles together, yielding a double result, just as would occur for built-in types and unlike what multiply does), but I'll leave that aspect of the problem in the form of an exercise for the reader [5].

Getting back to the main idea of this article, the transformation from a template class with non-template member function(s) to a non-template class with template member function(s) is worth considering every time you think about writing a class template, but it's not workable all the time. STL functor classes like multiplies have no state, hence they have no need to declare member variables of the types of the template's arguments. The situation is quite different for functor classes like binder2nd (the type of object returned by bind2nd). Here's the top of the interface for binder2nd, copied straight out of section 20.3.6.3 of the Standard:

```
template <class Operation>
class binder2nd
  : public unary_function<
      typename Operation::first_argument_type,
      typename Operation::result_type> {

protected:
  Operation op;
  typename Operation::second_argument_type value;

  ...
};
```

As you can see, objects of type binder2nd have two data members, and the type of each depends on binder2nd's template parameter, Operation. Because the types of these data members must be known before any binder2nd member functions can be called, there is no way to deduce these types via binder2nd member functions. (They *can* be deduced by non-member function templates, of course. That's precisely what bind2nd does!)

Summary

Sometimes you need class templates, sometimes you need function templates, and sometimes you need both. Anytime you find yourself writing a class template and giving it a non-template member function (especially operator()), you should ask yourself whether it might be better to make the member function the template and the class the non-template. If so, the best design of all may be one that templatizes both class and member functions and that combines them by putting the templatized member functions in an explicit specialization of the class template.

Notes and References

[1] Scott Meyers, *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, Addison-Wesley, 2001, ISBN 0-201-74962-9. The DeleteObject functor is described in Item 7.

[2] Donovan Rebbechi sent me this idea after my inaugural presentation of NewMultiplies at *The* C++ Seminar in March 2002. The next *The* C++ Seminar will take place in October 2002. For details, surf to http://www.thecppseminar.com/.

[3] Andrei Alexandrescu, *Modern C++ Design*, Addison-Wesley, 2001.

[4] This code compiles, links, and runs using MetroWerks CodeWarrior 6.1 and also using the Borland C++ command line compiler 5.5.1. It compiles, but fails to link using Comeau C/C++ 4.3 BETA#1. It fails to compile using Microsoft Visual Studio .NET (presumably due to that compiler's lack of support for template templates) and it elicits an internal compiler error using the mingw32 distribution of g++ 2.95.2. All tests were performed under Windows 2000.

[5] The solution that Donovan Rebbechi sent me includes support for mixed-type operations.

About the Author

Scott Meyers, author of the three best-selling *Effective C++* books, is one of the world's foremost authorities on C++ software development. He conceived of *The* C++ Seminar (http://www.thecppseminar.com/) and the Workshops on C++ Template Programming (http://www.oonumerics.org/tmpw00/ and tmpw01/), is a member of the *Software Development* advisory board, and provides training and consulting services to clients worldwide. His current interests focus on domain- and language-independent principles for improving software quality. His web site is http://www.aristeia.com/.