This is a pre-publication draft of the article I wrote for the October 1999 issue of *Dr. Dobbs Journal*. "Pre-publication" means this is what I sent to *DDJ*, but it may not be exactly the same as what appeared in print, because *DDJ* and I typically make small changes during preparation of the final version of the article.

Implementing operator->* for Smart Pointers

by Scott Meyers

When I wrote *More Effective C++* in 1995, one of the topics I examined was smart pointers. As a result, I get a fair number of questions about them, and one of the most interesting recent questions came from Andrei Alexandrescu. He asked, "Shouldn't a really smart smart pointer overload operator->*? I've never seen it done." I hadn't seen it done, either, so I set out to do it. The result is instructive, I think, and for more than just operator->*; it also involves insights into interesting and useful applications of templates.

Review of operator->*

If you're like most programmers, you don't use operator->* on a regular basis, so before I explain how to implement this operator for smart pointers, let me take a moment to review the behavior of the built-in version.

Given a class C, a pointer pmf to a parameterless member function of C, and a pointer pc to a C object, the expression

```
(pc->*pmf)();                  // invoke the member function *pmf on *pc
```

invokes the member function pointed to by pmf on the object pointed to by pc. Here's an example:

```
class Wombat {                 // wombats are cute Australian marsupials
public:                        // that look something like dogs

  int dig();                   // return depth dug
  int sleep();                 // return time slept

};

typedef int (Wombat::*PWMF)();              // PWMF is a pointer to a
                                            // Wombat member function

Wombat *pw = new Wombat;

PWMF pmf = &Wombat::dig;                    // make pmf point to
                                            // Wombat::dig

(pw->*pmf)();                               // same as pw->dig();

pmf = &Wombat::sleep;                       // make pmf point to
                                            // Wombat::sleep

(pw->*pmf)();                               // same as pw->sleep();
```

As you can see, pointers to member functions behave similarly to pointers to regular functions; the syntax is just a little more complicated. By the way, the parentheses around `pc->*pmf` are necessary, because the compiler would interpret

```
pc->*pmf();                 // error!
```

as

```
pc->*(pmf());               // error!
```

Designing Support for `operator->*`

Like many operators, `operator->*` is binary: it takes two arguments. When implementing `operator->*` for smart pointers, the left argument is a smart pointer to an object of type `T`. The right argument is a pointer to a member function of class `T`. The only thing that can be done with the result of a call to `operator->*` is to hand it a parameter list for a function call, so the return type of `operator->*` must be something to which `operator()` (the function call operator) may be applied. `operator->*`'s return value represents a pending member function call, so I'll call the type of object returned from `operator->*` *PMFC*, "Pending Member Function Call".

Put the above together, and you get the following pseudocode:

```
class PMFC {                // "Pending Member Function Call"
public:
  ...
  return type operator()( parameters ) const;
  ...
};

template<typename T>        // template for smart ptrs-to-T
class SP {                  // supporting operator->*
public:
  ...
  const PMFC operator->*( return type (T::*pmf)( parameters ) ) const;
  ...
};
```

Note that because each `PMFC` object represents a pending call to the member function passed to `operator->*`, both the member function and `PMFC::operator()` expect the same list of parameters.

To simplify matters for a moment, I'll assume that `T`'s member functions never take any arguments. (I'll remove this restriction below.) That means we can refine the pseudocode above as follows:

```
class PMFC {
public:
  ...
  return type operator()() const;
  ...
};

template<typename T>
class SP {
public:
  ...
```

```
    const PMFC operator->*( return type (T::*pmf)() ) const;
    ...
};
```

But what is the return type of the member function pointed to by `pmf`? It could be `int`, it could be `double`, it could be `const Wombat&`, it could be anything. We express this infinite set of possibilities in the usual fashion: we use a template. Hence, `operator->*` becomes a member function template. Furthermore, `PMFC` becomes a template, too, because different instantiations of `operator->*` must return different types of `PMFC` objects. (That's because each `PMFC` object must know what type to return when its `operator()` is invoked.)

After templatization, we can abandon pseudocode and write `PMFC` and `SP::operator->*` in C++. This is the result:

```
template<typename ReturnType>          // template for a pending mbr func
class PMFC {                           // call returning type ReturnType
public:
    ...
    ReturnType operator()() const;
    ...
};


template<typename T>
class SP {
public:
    ...
    template<typename ReturnType>
      const PMFC<ReturnType>
        operator->*( ReturnType (T::*pmf)() ) const;
    ...
};
```

Implementing `operator->*` for Zero-Parameter Member Functions

Let us now focus our attention on `PMFC`. `PMFC` represents a pending member function call, and that means it needs to know two things in order to implement its `operator()`: the member function to call and the object on which to invoke that member function. The `PMFC` constructor is the logical place to request these arguments. Furthermore, a standard `pair` object seems like a logical place to store them. That suggests this implementation:

```
template<typename ObjectType,            // class offering the mem func
          typename ReturnType,           // return type of the mem func
          typename MemFuncPtrType>       // full signature of the mem func
class PMFC {
public:
    typedef std::pair<ObjectType*, MemFuncPtrType> CallInfo;

    PMFC(const CallInfo& info): callInfo(info) {}

    ReturnType operator()() const
      { return (callInfo.first->*callInfo.second)(); }

private:
    CallInfo callInfo;
};
```

Though it may not look it at first glance, it's all pretty simple. When you create a PMFC, you specify which member function to call and which object on which to invoke it. When you later invoke the PMFC's operator() function, it just invokes the saved member function on the saved object.

Note how operator() is implemented in terms of the built-in operator->*. Because PMFC objects are created only when a smart pointer's user-defined operator->* is called, that means that user-defined operator->*s are implemented in terms of the built-in operator->*. This provides nice symmetry with the behavior of the user-defined operator-> with respect to that of the built-in operator->, because every call to a user-defined operator-> in C++ ultimately ends in an (implicit) call to the built-in operator->. Such symmetry is reassuring. It suggests that our design is on the right track.

You may have noticed that the template parameters ObjectType, ReturnType and MemFuncPtrType are somewhat redundant. Given MemFuncPtrType, it should be possible to figure out ObjectType and ReturnType. After all, both ObjectType and ReturnType are part of MemFuncPtrType. It is possible to deduce ObjectType and ReturnType from MemFuncPtrType using partial template specialization, but, because support for partial specialization is not yet common in commercial compilers, I've chosen not to use that approach in this article. For information on a design based on partial specialization, see the accompanying sidebar.

Given the above implementation of PMFC, SP<T>'s operator->* almost writes itself. The PMFC object it returns demands an object pointer and a member function pointer. Smart pointers conventionally store an object pointer, and the necessary member function pointer is just the parameter passed to operator->*. Thus:

```
template <typename T>
class SP {
public:
  SP(T *p): ptr(p) {}

  template <typename ReturnType>
    const PMFC<T, ReturnType, ReturnType (T::*)()>
      operator->*(ReturnType (T::*pmf)()) const
        { return std::make_pair(ptr, pmf); }

  ...

private:
  T* ptr;
};
```

That means that the following should work, and for the compilers with which I tested it (Visual C++ 6 and egcs 1.1.2), it does:

```
#include <iostream>
#include <utility>
using namespace std;

template<typename ObjectType,
         typename ReturnType,
         typename MemFuncPtrType>
class PMFC { ... };                        // as above

template <typename T>                      // also as above
class SP { ... };
```

```
class Wombat {
public:
  int dig()
  {
    cout << "Digging..." << endl;
    return 1;
  }
  int sleep()
  {
    cout << "Sleeping..." << endl;
    return 5;
  }
};

int main()
{                                          // as before, PWMF is a
  typedef int (Wombat::*PWMF)();           // pointer to a Wombat
                                           // member function

  SP<Wombat> pw = new Wombat;

  PWMF pmf = &Wombat::dig;                 // make pmf point to
                                           // Wombat::dig

  (pw->*pmf)();                            // invokes our operator->*;
                                           // prints "Digging..."

  pmf = &Wombat::sleep;                    // make pmf point to
                                           // Wombat::sleep

  (pw->*pmf)();                            // invokes our operator->*;
}                                          // prints "Sleeping..."
```

Yes, I know, this code has a resource leak (the `new`ed `Wombat` is never `delete`d) and it employs a using directive ("`using namespace std;`") when using declarations will do, but please try to focus on the interaction of `SP::operator->*` and `PMFC` instead of such relative minutiae. If you understand why the statements `(pw->*pmf)()` behave the way they do, there's no doubt you can easily fix the stylistic shortcomings of this example.

By the way, because both the `operator->*` member functions and all the `PMFC` member functions are (implicitly) inline, we may hope that the generated code for the statement

```
(pw->*pmf)();
```

using `SP` and `PMFC` will be the same as the generated code for the equivalent

```
(pw.ptr->*pmf)();
```

which uses only built-in operations. The runtime cost of using `SP`'s overloaded `operator->*` and `PMFC`'s overloaded `operator()` could thus be *zero*: zero additional bytes of code, zero additional bytes of data. The actual cost, of course, depends on the optimization capabilities of your compiler as well as on your standard library's implementation of `pair` and `make_pair`. For the two compilers (and associated libraries) with which I tested (after enabling full optimization), one yielded a zero-runtime-cost implementation of `operator->*`, but the other did not.

Adding Support for `const` Member Functions

Look closely at the formal parameter taken by `SP<T>`'s `operator->*` functions: it's `ReturnType (T::*pmf)()`. More specifically, it's *not* `ReturnType (T::*pmf)() const`. That means no pointer to a const member function can be passed to operator->*, and that means that operator->* fails to support const member functions. Such blatant const discrimination has no place in a well-designed software system. Fortunately, it's easy to eliminate. Simply add a second operator->* template to SP, one designed to work with pointers to const member functions:

```
template <typename T>
class SP {
public:
  ...                                                     // as above

  template <typename ReturnType>
    const PMFC<T, ReturnType, ReturnType (T::*)() const>  // const added
      operator->*(ReturnType (T::*pmf)() const) const     // const added
        { return std::make_pair(ptr, pmf); }

  ...                                                     // as above
};
```

Interestingly, there's no need to change anything in PMFC. Its type parameter MemFuncPtrType will bind to any type of member function pointer, regardless of whether the function in question is const.

Adding Support for Member Functions Taking Parameters

With the zero-parameter case under our belt, let's move on to support for pointers to member functions taking one parameter. The step is surprisingly small, because all we need to do is modify the type of the member-pointer parameter taken by operator->*, then propagate this change through PMFC. In fact, all we really need to do is add a new template parameter to operator->* (for the type of the parameter taken by the pointed-to member function), then update everything else to be consistent. Furthermore, because SP<T> should support member functions taking zero parameters as well as member functions taking one parameter, we *add* a new operator->* template to the existing one. In the code below, I show only support for non-const member functions, but operator->* templates for const member functions should be available, too.

```
template < typename ObjectType,
           typename ReturnType,
           typename MemFuncPtrType>
class PMFC {
public:
  typedef pair<ObjectType*, MemFuncPtrType> CallInfo;

  PMFC(const CallInfo& info)
  : callInfo(info) {}

  // support for 0 parameters
  ReturnType operator()() const
    { return (callInfo.first->*callInfo.second)(); }

  // support for 1 parameter
  template <typename Param1Type>
    ReturnType operator()(Param1Type p1) const
      { return (callInfo.first->*callInfo.second)(p1); }

private:
  CallInfo callInfo;
};
```

```
template <typename T>
class SP {
public:
  SP(T *p): ptr(p) {}

  // support for 0 parameters
  template <typename ReturnType>
    const PMFC<T, ReturnType, ReturnType (T::*)()>
      operator->*(ReturnType (T::*pmf)()) const
        { return std::make_pair(ptr, pmf); }

  // support for 1 parameter
  template <typename ReturnType,
            typename Param1Type>
    const PMFC<T, ReturnType, ReturnType (T::*)(Param1Type)>
      operator->*(ReturnType (T::*pmf)(Param1Type)) const
        { return std::make_pair(ptr, pmf); }

  ...

private:
  T* ptr;
};
```

Once you've got the hang of implementing support for 0 and 1 parameters, it's easy to add support for as many as you need. To support member functions taking *n* parameters, declare two member template `operator->*`s inside `SP`, one to support non-`const` member functions, one to support `const` ones. Each `operator->*` template should take *n*+1 type parameters, *n* for the parameters and one for the return type. Add the corresponding `operator()` template to `PMFC`, and you're done. You can find source code for `operator->*`s taking up to two parameters (supporting both `const` and non-`const` member functions) at the DDJ web site. [Editor: please provide appropriate details here.]

Packaging Support for `operator->*`

Many applications have several varieties of smart pointer[1], and it would be unpleasant to have to repeat the foregoing work for each one. Fortunately, support for `operator->*` can be packaged in the form of a base class:

```
template <typename T>       // base class for smart pointers wishing
class SmartPtrBase {        // to support operator->*
public:
  SmartPtrBase(T *initVal): ptr(initVal) {}

  // support for 0 parameters
  template <typename ReturnType>
    const PMFC<T, ReturnType, ReturnType (T::*)()>
      operator->*(ReturnType (T::*pmf)()) const
        { return std::make_pair(ptr, pmf); }

  // support for 1 parameter
  template <typename ReturnType,
            typename Param1Type>
    const PMFC<T, ReturnType, ReturnType (T::*)(Param1Type)>
      operator->*(ReturnType (T::*pmf)(Param1Type)) const
        { return make_pair(ptr, pmf); }
```

1. For an example of the different varieties of smart pointers that can be imagined (plus some killer-cool C++), check out Kevin S. Van Horn's web site, http://www.xmission.com/~ksvhsoft/code/smart_ptrs.html.

```
  ...
protected:
  T* ptr;
};
```

Smart pointers that wish to offer operator->* can then just inherit from SmartPtrBase.[2] However, it's probably best to use *private* inheritance, because the use of public inheritance would suggest the need to add a virtual destructor to SmartPtrBase, thus increasing its size (as well as the size of all derived classes). Private inheritance avoids this size penalty, though it mandates the use of a using declaration to make the privately inherited operator->* templates public:

```
template <typename T>
class SP: private SmartPtrBase<T> {
public:
  SP(T *p ): SmartPtrBase<T>(p) {}

  using SmartPtrBase<T>::operator->*;      // make the privately inherited
                                           // operator->* templates public

  // normal smart pointer functions would go here;  operator->*
  // functionality is inherited

};
```

To package things even more nicely, both SmartPtrBase and the PMFC template could be put in a namespace.

Loose Ends

After I'd developed this approach to implementing operator->* for smart pointers, I posted my solution to the Usenet newsgroup comp.lang.c++.moderated to see what I'd overlooked. It wasn't long before Esa Pulkkinen made these observations:

> There are at least two problems with your approach:
>
> 1. You can't use pointers to data members (though this is easy enough to solve).
>
> 2. You can't use user-defined pointers-to-members. If someone has overloaded operator->* to take objects that act like member pointers, you may want to support such "smart pointers to members" in your smart pointer class. Unfortunately, you need traits classes to get the result type of such overloaded operator->*s.

Smart pointers to members! Yikes! Esa's right.[3] Fortunately, this article is long enough that I can stop here and leave ways of addressing Esa's observations in the time-honored form of exercises for the reader. So I will.

---

2. This design applies only to smart pointers that contain dumb pointers to do the actual pointing. This is the most common smart pointer design, but there are alternatives. Such alternative designs may need to package operator->* functionality in a manner other than that described here.

3. He's righter than I originally realized. Shortly after writing the draft of this article, one of my consulting clients showed me a problem that was naturally solved by smart pointers to members. I was surprised, too.

Summary

If your goal is to make your smart pointers as behaviorally compatible with built-in pointers as possible, you should support `operator->*`, just like built-in pointers do. The use of class and member templates makes it easy to implement such support, and packaging the implementation in the form of a base class facilitates its reuse by other smart pointer authors.

Acknowledgements

In addition to motivating my interest in `operator->*` in the first place, Andrei Alexandrescu helped me simplify my implementation of PMFC. Andrei also provided insightful comments on earlier drafts of this paper and the accompanying source code, as did Esa Pulkkinen and Mark Rodgers. I am greatly indebted to each of them for their considerable help with this article.

About the Author

Scott Meyers is an internationally recognized author and consultant on C++ software development. He spent most of 1998 working on *Effective C++ CD*, an innovative HTML-based version of his best selling books, *Effective C++* and *More Effective C++*. You can find out more about Scott at his home page, http://www.aristeia.com/.

Partial Template Specialization and `operator->*` [Sidebar]

As I worked on this article, Esa Pulkkinen and Mark Rodgers pointed out that partial template specialization can be used to extract the object and return type of a member function from the type of a pointer to that member function. One need merely apply the traits[4] technique (a technique used widely in the standard C++ library).

Mark Rodgers suggested the following implementation for member functions taking zero or one parameters. (The extension to more parameters is straightforward.)

```
template <typename T>                              // traits class
struct MemFuncTraits {};

template <typename R, typename O>                  // partial specialization
struct MemFuncTraits<R (O::*)()> {                 // for zero-parameter
  typedef R ReturnType;                            // non-const member
  typedef O ObjectType;                            // functions
};

template <typename R, typename O>                  // partial specialization
struct MemFuncTraits<R (O::*)() const> {           // for zero-parameter
  typedef R ReturnType;                            // const member
  typedef O ObjectType;                            // functions
};

template <typename R, typename O, typename P1>     // partial specialization
struct MemFuncTraits<R (O::*)(P1)> {               // for one-parameter
  typedef R ReturnType;                            // non-const member
  typedef O ObjectType;                            // functions
};

template <typename R, typename O, typename P1>     // partial specialization
struct MemFuncTraits<R (O::*)(P1) const> {         // for one-parameter
  typedef R ReturnType;                            // const member
  typedef O ObjectType;                            // functions
};
```

Given these templates, `PMFC` can be simplified to take only one type parameter, `MemFuncPtrType`. That's because the other two type parameters — `ObjectType` and `ReturnType` — can be deduced from `MemFuncPtrType`:

- `ObjectType` is `MemFuncTraits<MemFuncPtrType>::ObjectType`
- `ReturnType` is `MemFuncTraits<MemFuncPtrType>::ReturnType`

That leads to this revised implementation of `PMFC`:

```
template <typename MemFuncPtrType>
class PMFC {
public:

  typedef typename MemFuncTraits<MemFuncPtrType>::ObjectType ObjectType;

  typedef typename MemFuncTraits<MemFuncPtrType>::ReturnType ReturnType;

  ...                               // same as before

};
```

4. See Nathan Myers' article, "Traits: A New and Useful Template Technique," originally published in the June 1995 *C++ Report* and now available at `http://www.cantrip.org/traits.html`.

Other than offering a chance to show off our knowledge of traits and when `typename` must precede the name of a type in a template, this doesn't appear to have bought us much, but don't be fooled; this greatly reduces the work smart pointer classes must do to implement `operator->*`. In fact, Mark Rodgers noted that *a single* `operator->*` *template* can support *all possible member function pointers*, regardless of the number of parameters taken by the member functions and whether the member functions are `const`. Just replace all the `operator->*` templates in `SP` (or `SmartPtrBase`) with this:

```
template <typename MemFuncPtrType>
  const PMFC<MemFuncPtrType>
    operator->*(MemFuncPtrType pmf) const
      { return std::make_pair(ptr, pmf); }
```

The type parameter `MemFuncPtrType` will bind to *any* pointer to member function type, regardless of parameters, return type, and constness. It will then pass that type on to `PMFC`, where partial specialization will be used to pick the type apart.

You can find source code employing this approach to implementing `operator->*` at the DDJ web site. [Editor: please provide appropriate details here.]