This is a pre-publication draft of the column I wrote for the November-December 1996 issue of the *C++ Report*. "Pre-publication" means this is what I sent to the *Report*, but it may not be exactly the same as what appeared in print, because the *Report* and I often make small changes after I submit the "final" draft for a column. Comments? Feel free to send me mail: `smeyers@aristeia.com`.

Refinements to Smart Pointers

My last three columns described the design and implementation of smart pointers, with an emphasis on getting them to look as much as possible like dumb pointers. Since publication of those columns, I've been told of some refinements to my design that make smart pointers look even more like dumb pointers. In this column — my last for the *C++ Report* — I discuss these improvements.

Nullness Testing and Mixed-type Comparisons

In June, I examined the problem of how to test smart pointers for nullness. I showed how the "natural" conversion from a smart pointer to `void*` or to `bool` allowed mixed-type comparisons that would not be allowed for dumb pointers:

```
template<class T>
class SmartPtr {
public:
  ...
  operator bool();              // returns true iff the
  ...                           // smart ptr is non-null
};

SmartPtr<Apple> pa;
SmartPtr<Orange> po;

...

if (pa == po) ...       // unfortunately, this compiles due
                        // to implicit conversions to bool
```

To avoid such problems, I suggested providing only `operator!`:

```
template<class T>
class SmartPtr {
public:
  ...
  bool operator!() const;       // returns true iff the
  ...                           // smart ptr is null

};
```

This lets clients program like this,

```
SmartPtr<TreeNode> ptn;

...

if (!ptn) ...                   // fine
```

but not like this:

```
if (ptn == 0) ...               // error!

if (ptn) ...                    // also an error
```

In response to this column, readers Eric Hopper and Michael Klobe each sent mail showing how they'd come up with different solutions to the problem. Eric wrote:

> Providing an `operator bool()` conversion doesn't necessarily lead to strange conversions being applied willy-nilly. Many of these conversions can be eliminated by the judicious use of a base class. Of course, base classes and inheritance are currently unfashionable concepts, so you may not want to use them, but here is how you might do it:

```
class SmartPtrBase {
private:
  // No implementations needed for the following, a la
  // hidden copy and assignment functions
  void operator==(const SmartPtrBase &b) const;
  void operator!=(const SmartPtrBase &b) const;
  ...
};

template<class T>
class SmartPtr: public SmartPtrBase {
public:
  operator bool() const { return(ptr != 0); }

  bool operator==(const SmartPtr<T> &b) const
  { return(ptr == b.ptr); }

  ...

private:
  T *ptr;
};

SmartPtr<Fred> a1;
SmartPtr<Fred> a2;

...

if (a1 == a2) ...   // fine

SmartPtr<Joe> b;

...

if (a1 == b) ...    // error, operator== is private
```

In other words, all smart pointer classes inherit from `SmartPtrBase`, and `SmartPtrBase` declares the comparison operators `private`. This makes any kind of smart pointer comparison illegal. Derived classes then define their own homogeneous comparison operators if they wish to support comparisons of smart pointers. Very nice.

I can think of two drawbacks to this approach. First, it requires that all smart pointer classes inherit from `SmartPtrBase`. Some programmers rebel against mandatory base classes for small, light-weight objects like smart pointers. However, `SmartPtrBase` should have no data, so it shouldn't affect the size or performance of derived class objects in any way.

It's worth noting that the use of public inheritance implies that `SmartPtrBase` should have a virtual destructor, and that could add a vptr to smart pointer classes that might otherwise not have one. That problem can be eliminated by using private inheritance between `SmartPtr` and `SmartPtrBase`. Using private inheritance would make it necessary to use a cast to convert from a `SmartPtr<T>` to a `SmartPtrBase` (i.e., from a typed smart pointer to an untyped smart pointer), and this is inconsistent with the behavior of dumb pointers (which may be implicitly converted to `void*`s). But a `SmartPtrBase` object, like a `void*`, is essentially useless until cast back to the correct type. If type-safe casts (i.e., `dynamic_cast`s) are to be supported on `SmartPtrBase` objects, `SmartPtrBase` must have at least one virtual function (in which case its derived classes must accept a vptr), and public inheritance is appropriate. If

dynamic_casts are not needed, private inheritance and a nonvirtual destructor strikes me as a reasonable design.

The second drawback to Eric's approach has to do with mixed smart-and-dumb comparisons. Consider this code:

```
SmartPtr<Apple> spa = new Apple;  // spa = smart ptr to Apple
Apple *dpa = new Apple;           // dpa = dumb ptr to Apple
if (spa == dpa) ...               // should this compile?
```

It doesn't seem unreasonable to expect the comparison to compile, and in fact it will (unless the smart pointer constructor taking a dumb pointer is declared `explicit`[1]). It does so, however, via implicit conversion of `dpa` from a dumb pointer to a smart pointer via the smart pointer constructor. In other words, the code is treated like this:

```
if (spa == static_cast< SmartPtr<Apple> >(dpa)) ...
```

However, if this compiles,

```
if (spa == dpa) ...       // compare smart ptr to dumb ptr
```

surely this should compile as well:

```
if (dpa == spa) ...       // compare dumb ptr to smart ptr
```

Unless `SmartPtr<T>` provides an `operator T*` function, however, it will not, and my June column discussed reasons why such a conversion is generally undesirable. That being the case, making `operator==` (or any other relational operator) a member function is generally inferior to declaring it as a non-member. I'd thus modify Eric's design by declaring a non-member template to handle homogeneous comparisons:

```
class SmartPtrBase { ... };  // as above

template<class T>
class SmartPtr:
  private SmartPtrBase {     // note private inheritance
public:
  operator bool() const { return(ptr != 0); }

  ...

friend bool operator==(const SmartPtr<T>& p1,
                       const SmartPtr<T>& p2);
};

template<class T>
inline bool operator==(const SmartPtr<T>& p1,
                       const SmartPtr<T>& p2)
{ return p1.ptr == p2.ptr; }

SmartPtr<Fred> a1;
SmartPtr<Joe> b;

...

if (a1 == b) ...   // error, operator== is still private

SmartPtr<Fred> a2;

if (a1 == a2) ...  // okay, instantiates operator==<Fred>

Fred *dpf = new Fred;

if (a1 == dpf) ... // these comparisons fail to compile
if (dpf == a1) ...
```

Note that now *both* smart-and-dumb pointer comparisons fail to compile. It would be preferable if we could somehow make them both work, but consider what would be re-

1. Scott Meyers, *More Effective C++*, Addison-Wesley, 1996, pp. 24-31.

quired for that to happen. First the compiler would have to instantiate `operator==<Fred>`, then it would have to apply a user-defined conversion to the dumb pointer to turn it into a `SmartPtr<Fred>` object. The rules of C++ don't allow compilers to do that, because the combination of implicit template instantiation and implicit type conversion via user-defined functions could too easily allow function calls to succeed when programmers would expect them to fail.

With the non-member `operator==` template, both

```
if (a1 == dpf) ...     // error!
if (dpf == a1) ...     // error!
```

will be treated the same way; that's a characteristic Eric's solution lacks.

Michael Klobe suggested a different solution, one that avoids the use of (unfashionable?) base classes:

> I tend to use the conversion-to-`bool` technique, and introduce two equality operators to prevent the mixed-type comparisons:
>
> ```
> template< class T >
> class smart {
> public:
>   // Constructor.
>   smart( T* p = 0 ): pointer_( p ) {}
>
>   // Conversion operator to ease nullness testing
>   operator bool() { return pointer_; }
>
>   // Equality comparison
>   bool operator==( const smart< T >& s ) const
>   { return pointer_ == s.pointer_; }
>
>   ...
>
> private:
>   T* pointer_;
> };
>
> // Redundant equality operator to prevent
> // smart< X > == smart< Y >. A similar technique could be
> // used for operator< and other needed relational
> // operators.
> template< class T1, class T2 >
> inline bool
> operator==( const smart< T1 >& s1, const smart< T2 >& s2 )
> { return s1.operator==( s2 ); }
>
> smart< int > pint1;
> smart< int > pint2;
>
> if ( pint1 == pint2 ) ...      // compiles as expected
>
> smart< long > plong;
>
> if ( pint1 == plong ) ...      // won't compile!
> ```
>
> This will cause compile-time diagnostics for the mixed-type comparison.

Because Michael's solution is based on non-member functions, it handles mixed smart-and-dumb comparisons in a uniform manner: it rejects them.

Implicit Smart-to-Dumb Conversions and Smart Pointer Deletion

Attacking a different problem in a different forum, John Hickin demonstrated a clever way to eliminate the "particularly nasty bug" I described in my June column. I was considering whether it was wise to provide implicit smart-to-dumb pointer conver-

sions, and I explained how such conversions make it possible to delete smart pointer *objects*:

```
template<class T>              // template for smart ptrs
class DBPtr {                  // to database objects
public:
  ...
  operator T*() const { return pointee; }
  ...
private:
  T *pointee;
};
void processTuple(Tuple *pt);  // some function that takes
                               // a dumb Tuple* pointer
DBPtr<Tuple> pt = new Tuple;
...
processTuple(pt);              // fine, implicitly converts
                               // pt to a Tuple*
delete pt;                     // also fine -- converts pt
                               // to a Tuple*, then
                               // deletes that!
```

A related topic came up in the Usenet discussion group `comp.std.c++`, and I posted my observation there. John Hickin made a follow-up posting in which he off-handedly remarked that he solved this problem by adding a *second* smart-to-dumb pointer conversion to the smart pointer template: conversion to `void*`!

```
template<class T>
class DBPtr {
public:
  ...
  operator T*() const { return pointee; }
  ...
private:
  operator void*() const;      // this is new
  T *pointee;
};
DBPtr<Tuple> pt = new Tuple;
...
processTuple(pt);              // fine, still implicitly
                               // converts pt to a Tuple*
delete pt;                     // error! ambiguous call:
                               // convert to Tuple* or to
                               // void*?
```

I really like this solution, because it demonstrates something I don't recall having seen before: the deliberate introduction of ambiguity to prevent a function call from succeeding. Usually when we want function calls to fail, we either avoid declaring the function or (as in Eric's design above) we declare the function private. The algorithm for resolving calls to overloaded functions, however, may fail for any of three reasons:[2]

- There is no function to call

2. For more details on resolving calls to overloaded functions than any human should have to master, see Josee Lajoie's columns in the July/August 1995 and the January 1996 *C++ Report*s

- The called function is inaccessible
- The function call is ambiguous

John's design takes advantage of the oft-overlooked third option.

Further Smart Pointer Reading

If you're still interested in smart pointers, I encourage you to check out Cay Horstmann's 1993 article on the topic.[3] The ideas he presents there can be employed in myriad ways, so don't be put off by the fact that the topic at hand is memory management in DOS.

If you're *really* interested in smart pointers, consider Jeff Alger's book,[4] which has more to say about smart pointers than any other source I know of. In fact, Jeff is so sold on smart pointers, he offers this advice (I am not making this up):

- Use smart pointers, even if you aren't sure why you should.[5]

I can't generate quite as much enthusiasm about smart pointers as Alger does, nor could I bring myself (as he does) to call some smart pointers "insufferably brilliant," but perhaps that's just because I'm shallow :-) [6]

But Wait! — There's No More

For the longest time I was unable to fathom why so many newspaper and magazine columns are vacuous and insipid. Now I know. When a deadline approaches, you are required to come up with enough material to fill a particular number of column inches, and inspiration fails, you lower your standards until you can spit out something — *anything* — that will fill the allotted space. (This also explains why television tends to be atrocious: bad programming is preferable to no programming.)

I've been writing this column for over four years now, and I find myself running out of worthwhile things to say. Furthermore, having in the past six years also written one dissertation, two books, the bulk of two others, several technical papers, a few magazine articles, and countless email messages relating to the above, I'm tired of writing. I never intended to be an author, and I now wish to retire from it for a while. Instead, I want to turn my efforts to a different kind of authorship, a kind I've neglected for too long: *real* programming.

I've *studied* C++ for years, but now I want to start *using* it again. By the time you read this, I should have been back at it for several months, cursing compilers, fighting with debuggers, battling operating systems, and desperately trying to understand application frameworks and other APIs. Just like you. Frankly, I suspect that little has fundamentally changed since 1990, when I wrote a paper with the best title I ever coined: "Object-oriented programming: the view from the trenches is not always pretty."[7]

I'm looking forward to seeing that view again.

Acknowledgment

During preparation of this column, Josee Lajoie patiently explained the latest rules governing template instantiation and how they interact with implicit type conversions on function parameters.

---

3. Cay Horstmann, "Memory Management and Smart Pointers," *C++ Report*, March-April 1993.
4. Jeff Alger, *Secrets of the C++ Masters*, Academic Press, 1995.
5. *Ibid*, pg. 224.
6. Cay Horstmann, "New (PC) Products," *C++ Report*, April 1996.
7. "Working with Object-Oriented Programs: The View from the Trenches is Not Always Pretty," *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications* (SOOPPA), September 1990.

Author Bio

Scott Meyers has a Ph.D. in Computer Science and is the author of *Effective C++* and *More Effective C++*; he is also primary author of *The Downloader's Companion for Windows* and *The Downloader's Companion for Windows 95*. He provides C++ training and consulting services to clients worldwide. Scott can be reached via email at `smeyers@aristeia.com`.