This is a pre-publication draft of the column I wrote for the June 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`.

Smart pointers, Part 2

In my last column, we began an examination of smart pointers by seeing how they are declared and implemented, focusing on the topics of construction, destruction, assignment, dereferencing, and ownership of what is pointed to. In this installment, we begin to look more closely at the other operations supported by dumb pointers, and we see how well we can support those operations in smart pointer classes.

Testing Smart Pointers for Nullness

With the functions we have discussed so far, we can create, destroy, copy, assign, and dereference smart pointers. One of the things we cannot do, however, is find out if a smart pointer is null:

```
SmartPtr<TreeNode> ptn;

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

This is a serious limitation.

It would be easy to add an `isNull` member function to our smart pointer classes, but that would lead to the problem that smart pointers don't act like dumb pointers when testing for nullness. A different approach is to provide an implicit conversion operator that allows the tests above to compile. The conversion traditionally employed for this purpose is to `void*`:

```
template<class T>
class SmartPtr {
public:
  ...
  operator void*();            // returns 0 if the smart
  ...                          // ptr is null, nonzero
};                             // otherwise

SmartPtr<TreeNode> ptn;

...
if (ptn == 0) ...              // now fine
if (ptn) ...                   // also fine
if (!ptn) ...                  // fine
```

This is similar to a conversion provided by the iostream classes, and it explains why it's possible to write code like this:

```
ifstream inputFile("datafile.dat");
```

```
if (inputFile) ...              // test to see if inputFile
                                // was successfully
                                // opened
```

Like all type conversion functions, this one has the drawback of letting function calls succeed that most programmers would expect to fail [cite 1]. In particular, it allows comparisons of smart pointers of completely different types:

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

...

if (pa == po) ...               // this compiles!
```

Even if there is no `operator==` taking a `SmartPtr<Apple>` and a `SmartPtr<Orange>`, this compiles, because both smart pointers can be implicitly converted into `void*` pointers, and there is a built-in comparison function for built-in pointers. This kind of behavior makes implicit conversion functions dangerous.

There are variations on the conversion-to-`void*` motif. Some designers advocate conversion to `const void*`, others embrace conversion to `bool`. Neither of these variations eliminates the problem of allowing mixed-type comparisons.

There is a middle ground that allows you to offer a reasonable syntactic form for testing for nullness while minimizing the chances of accidentally comparing smart pointers of different types. It is to overload `operator!` for your smart pointer classes so that `operator!` returns `true` if and only if the smart pointer on which it's invoked is null:

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

This lets your clients program like this,

```
SmartPtr<TreeNode> ptn;

...

if (!ptn) {                     // fine
  ...                           // ptn is null
}
else {
  ...                           // ptn is not null
}
```

but not like this:

```
if (ptn == 0) ...               // still an error

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

The only risk for mixed-type comparisons is statements such as these:

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

...

if (!pa == !po) ...             // alas, this compiles
```

Fortunately, programmers don't write code like this very often. Interestingly, iostream library implementations provide an `operator!` in addition to the implicit conversion to `void*`, but these two functions typically test for slightly different stream

states. (In the C++ library standard, the implicit conversion to `void*` has been replaced by an implicit conversion to `bool`, and `operator bool` always returns the negation of `operator!`.)

Converting Smart Pointers to Dumb Pointers

Sometimes you'd like to add smart pointers to an application or library that already uses dumb pointers. For example, your distributed database system (from my last column) may not originally have been distributed, so you may have some old library functions that aren't designed to use smart pointers:

```
class Tuple {                   // class for database
public:                         // tuples
  ...
};

void normalize(Tuple *pt);      // put *pt into canonical
                                // form; note use of dumb
                                // pointer
```

Consider what will happen if you try to call `normalize` with a smart pointer-to-`Tuple`:

```
DBPtr<Tuple> pt;

...

normalize(pt);                  // error!
```

The call will fail to compile, because there is no way to convert a `DBPtr<Tuple>` to a `Tuple*`. You can make it work by doing this,

```
normalize(&*pt);                // gross, but legal
```

but I hope you'll agree this is repugnant.

The call can be made to succeed by adding to the smart pointer-to-T template an implicit conversion operator to a dumb pointer-to-T:

```
template<class T>               // as before
class DBPtr {
public:
  ...
  operator T*() { return pointee; }
  ...
};

DBPtr<Tuple> pt;

...

normalize(pt);                  // this now works
```

Addition of this function also eliminates the problem of testing for nullness:

```
if (pt == 0) ...                // fine, converts pt to a
                                // Tuple*

if (pt) ...                     // ditto

if (!pt) ...                    // ditto (reprise)
```

However, there is a dark side to such conversion functions. They make it easy for clients to program directly with dumb pointers, thus bypassing the smarts your pointer-like objects are designed to provide:

```
void processTuple(DBPtr<Tuple>& pt)
{
  Tuple *rawTuplePtr = pt;      // converts DBPtr<Tuple> to
                                // Tuple*
```

```
  use rawTuplePtr to modify the tuple;
}
```

Usually, the "smart" behavior provided by a smart pointer is an essential component of your design, so allowing clients to use dumb pointers typically leads to disaster. For example, if `DBPtr` implements reference-counting, allowing clients to manipulate dumb pointers directly will almost certainly lead to bookkeeping errors that corrupt the reference-counting data structures.

Even if you provide an implicit conversion operator to go from a smart pointer to the dumb pointer it's built on, your smart pointer will never be truly interchangeable with the dumb pointer. That's because the conversion from a smart pointer to a dumb pointer is a user-defined conversion, and compilers are forbidden from applying more than one such conversion at a time. For example, suppose you have a class representing all the clients who have accessed a particular tuple:

```
class TupleAccessors {
public:
  TupleAccessors(const Tuple *pt);  // pt identifies the
  ...                               // tuple whose accessors
};                                  // we care about
```

As usual, `TupleAccessors`' single-argument constructor also acts as a type-conversion operator from `Tuple*` to `TupleAccessors`. Now consider a function for merging the information in two `TupleAccessors` objects:

```
TupleAccessors merge(const TupleAccessors& ta1,
                     const TupleAccessors& ta2);
```

Because a `Tuple*` may be implicitly converted to a `TupleAccessors`, calling `merge` with two dumb `Tuple*` pointers is fine:

```
Tuple *pt1, *pt2;

...

merge(pt1, pt2);    // fine, both pointers are converted
                    // to TupleAccessors objects
```

The corresponding call with smart `DBPtr<Tuple>` pointers, however, fails to compile:

```
DBPtr<Tuple> pt1, pt2;

...

merge(pt1, pt2);    // error! No way to convert pt1 and
                    // pt2 to TupleAccessors objects
```

That's because a conversion from `DBPtr<Tuple>` to `TupleAccessors` calls for *two* user-defined conversions (one from `DBPtr<Tuple>` to `Tuple*` and one from `Tuple*` to `TupleAccessors`), and such sequences of conversions are prohibited by the language.

Smart pointer classes that provide an implicit conversion to a dumb pointer open the door to a particularly nasty bug. Consider this code:

```
DBPtr<Tuple> pt = new Tuple;

...

delete pt;
```

This should not compile. After all, `pt` is not a pointer, it's an object, and you can't delete an object. Only pointers can be deleted, right?

Right. But remember that compilers use implicit type conversions to make function calls succeed whenever they can, and recall that use of the `delete` operator leads to

calls to a destructor and to `operator delete`, both of which are functions. Compilers want these function calls to succeed, so in the `delete` statement above, they implicitly convert `pt` to a `Tuple*`, then they delete that. This will almost certainly break your program.

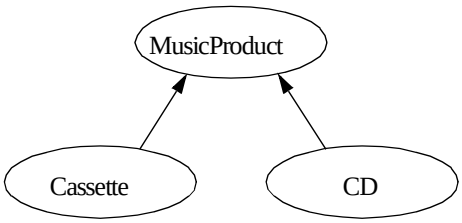
If `pt` owns the object it points to, that object is now deleted twice, once at the point where `delete` is called, a second time when `pt`'s destructor is invoked. If `pt` doesn't own the object, somebody else does. That somebody may be the person who deleted `pt`, in which case all is well. If, however, the owner of the object pointed to by `pt` is not the person who deleted `pt`, we can expect the rightful owner to delete that object again later. The first and last of these scenarios leads to an object being deleted twice, and deleting an object more than once yields undefined behavior.

This bug is especially pernicious because the whole idea behind smart pointers is to make them look and feel as much like dumb pointers as possible. The closer you get to this ideal, the more likely your clients are to forget they are using smart pointers. If they do, who can blame them if they continue to think that in order to avoid resource leaks, they must call `delete` if they called `new`?

The bottom line is simple: don't provide implicit conversion operators to dumb pointers unless there is a compelling reason to do so.

Smart Pointers and Inheritance-Based Type Conversions

Suppose we have a public inheritance hierarchy modeling consumer products for storing music:




```
class MusicProduct {
public:
  MusicProduct(const string& title);
  virtual void play() const = 0;
  virtual void displayTitle() const = 0;
  ...
};

class Cassette: public MusicProduct {
public:
  Cassette(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};

class CD: public MusicProduct {
public:
  CD(const string& title);
  virtual void play() const;
  virtual void displayTitle() const;
  ...
};
```

Further suppose we have a function that, given a `MusicProduct` object, displays the title of the product and then plays it:

```
void displayAndPlay(const MusicProduct* pmp, int numTimes)
{
  for (int i = 1; i <= numTimes; ++i) {
    pmp->displayTitle();
    pmp->play();
  }
}
```

Such a function might be used like this:

```
Cassette *funMusic = new Cassette("Alapalooza");
CD *nightmareMusic = new CD("Disco Hits of the 70s");

displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 0);
```

There are no surprises here, but look what happens if we replace the dumb pointers with their allegedly smart counterparts:

```
void displayAndPlay(const SmartPtr<MusicProduct>& pmp,
                    int numTimes);

SmartPtr<Cassette> funMusic(new Cassette("Alapalooza"));
SmartPtr<CD> nightmareMusic(new CD("Disco Hits of the 70s"));

displayAndPlay(funMusic, 10);          // error!
displayAndPlay(nightmareMusic, 0);     // error!
```

If smart pointers are so brainy, why won't these compile?

They won't compile because there is no conversion from a `SmartPtr<CD>` or a `SmartPtr<Cassette>` to a `SmartPtr<MusicProduct>`. As far as compilers are concerned, these are three separate classes — they have no relationship to one another. Why should compilers think otherwise? After all, it's not like `SmartPtr<CD>` or `SmartPtr<Cassette>` inherits from `SmartPtr<MusicProduct>`. With no inheritance relationship between these classes, we can hardly expect compilers to run around converting objects of one type to objects of other types.

Fortunately, there is a way to get around this limitation, and the idea (if not the practice) is simple: give each smart pointer class an implicit type conversion operator for each smart pointer class to which it should be implicitly convertible. For example, in the music hierarchy, you'd add an `operator SmartPtr<MusicProduct>` to the smart pointer classes for `Cassette` and `CD`:

```
class SmartPtr<Cassette> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
  Cassette *pointee;
};

class SmartPtr<CD> {
public:
  operator SmartPtr<MusicProduct>()
  { return SmartPtr<MusicProduct>(pointee); }

  ...

private:
  CD *pointee;
};
```

The drawbacks to this approach are twofold. First, you must manually specialize the `SmartPtr` class instantiations so you can add the necessary implicit type conversion operators, but that pretty much defeats the purpose of templates. Second, you may

have to add many such conversion operators, because your pointed-to object may be deep in an inheritance hierarchy, and you must provide a conversion operator for *each* base class from which that object directly or indirectly inherits. (If you think you can get around this by providing only an implicit type conversion operator for each direct base class, think again. Because compilers are prohibited from employing more than one user-defined type conversion function at a time, they can't convert a smart pointer-to-`T` to a smart pointer-to-indirect-base-class-of-`T` unless they can do it in a single step.)

It would be quite the time-saver if you could somehow get compilers to write all these implicit type conversion functions for you. Thanks to a recent language extension (member templates), you can. We'll examine member templates and how they solve this problem *almost* perfectly in my next column, where we'll finish our examination of smart pointers.

References

1. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996, pp. 24-31.

Author Bio

Scott Meyers has a Ph.D. in Computer Science and is the author of *Effective C++* and *More Effective C++*; he provides C++ consulting services to clients worldwide. This column is based on material in *More Effective C++*. Scott can be reached via email at `smeyers@aristeia.com`.