This is a pre-publication draft of the column I wrote for the September 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 3

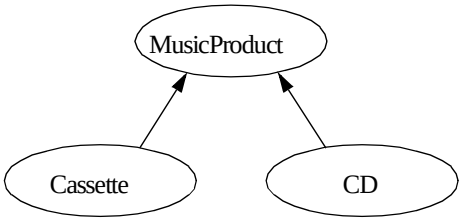
We ended my last column considering the problem of smart pointers and inheritance-based type conversions. We discovered that because smart pointer classes are generated from templates, there are no inheritance relationships between smart pointers, even if the dumb pointers they emulate are related by inheritance. However, we also found that we could approximate dumb pointer inheritance by adding implicit type conversion operators to smart pointer classes, and I promised I'd show in this column how the use of member function templates (usually just called *member templates*) can get your compilers to generate the implicit type conversion operators you need.

You do it like this:

```
template<class T>              // template class for smart
class SmartPtr {               // pointers-to-T objects
public:
  SmartPtr(T* realPtr = 0);

  T* operator->() const;
  T& operator*() const;

  template<class newType>      // template function for
  operator SmartPtr<newType>() // implicit conversion ops.
  {
    return SmartPtr<newType>(pointee);
  }

  ...
};
```

Now hold on to your headlights, this isn't magic — but it's close. It works as follows. (I'll give a specific example in a moment, so don't despair if the remainder of this paragraph reads like so much gobbledygook. After you've seen the example, it'll make more sense, I promise.) Suppose a compiler has a smart pointer-to-`T` object, and it's faced with the need to convert that object into a smart pointer-to-base-class-of-`T`. The compiler checks the class definition for `SmartPtr<T>` to see if the requisite conversion operator is declared, but it is not. (It can't be: no conversion operators are declared in the template above.) The compiler then checks to see if there's a member function template it can instantiate that would let it perform the conversion it's looking for. It finds such a template (the one taking the formal type parameter `newType`), so it instantiates the template with `newType` bound to the base class of `T` that's the target of the conversion. At that point, the only question is whether the code for the instantiated member function will compile. In order for it to compile, it must be legal to pass the (dumb) pointer `pointee` to the constructor for the smart pointer-to-base-of-`T`. `pointee` is of type `T`, so it is certainly legal to convert it into a pointer to its (public or protected) base classes. Hence, the code for the type conversion operator will compile, and the implicit conversion from smart pointer-to-`T` to smart pointer-to-base-of-`T` will succeed.

An example will help. Let us return to the music hierarchy of CDs, cassettes, and music products we considered in my last column:



We saw last time that the following code wouldn't compile, because there was no way for compilers to convert the smart pointers to CDs or cassettes into smart pointers to music products:

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

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

displayAndPlay(funMusic, 10);       // used to be an error
displayAndPlay(nightmareMusic, 0);  // used to be an error
```

With the revised smart pointer class containing the member function template for implicit type conversion operators, this code will succeed. To see why, look at this call:

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

The object `funMusic` is of type `SmartPtr<Cassette>`. The function `displayAndPlay` expects a `SmartPtr<MusicProduct>` object. Compilers detect the type mismatch and seek a way to convert `funMusic` into a `SmartPtr<MusicProduct>` object. They look for a single-argument constructor in the `SmartPtr<MusicProduct>` class that takes a `SmartPtr<Cassette>`, but they find none. They look for an implicit type conversion operator in the `SmartPtr<Cassette>` class that yields a `SmartPtr<MusicProduct>` class, but that search also fails. They then look for a member function template they can instantiate to yield one of these functions. They discover that the template inside `SmartPtr<Cassette>`, when instantiated with `newType` bound to `MusicProduct`, generates the necessary function. They instantiate the function, yielding the following code:

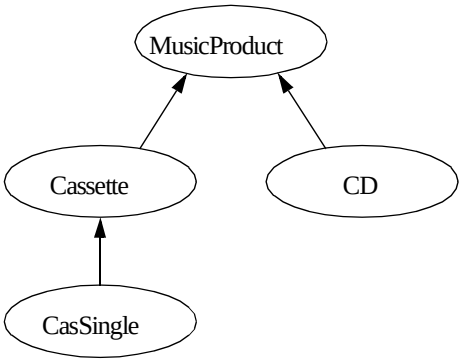
```
SmartPtr<Cassette>::operator SmartPtr<MusicProduct>()
{
  return SmartPtr<MusicProduct>(pointee);
}
```

Will this compile? For all intents and purposes, nothing is happening here except the calling of the `SmartPtr<MusicProduct>` constructor with `pointee` as its argument, so the real question is whether one can construct a `SmartPtr<MusicProduct>` object with a `Cassette*` pointer. The `SmartPtr<MusicProduct>` constructor expects a `MusicProduct*` pointer, but now we're on the familiar ground of conversions between dumb pointer types, and it's clear that `Cassette*` can be passed in where a `MusicProduct*` is expected. The construction of the `SmartPtr<MusicProduct>` is therefore successful, and the conversion of the `SmartPtr<Cassette>` to SmartPtr`<MusicProduct>` is equally successful. *Voilà!* Implicit conversion of smart pointer types. What could be simpler?

Furthermore, what could be more powerful? Don't be misled by this example into assuming that this works only for pointer conversions up an inheritance hierarchy. The method shown succeeds for *any* legal

implicit conversion between pointer types. If you've got a dumb pointer type `T1` and another dumb pointer type `T2`, you can implicitly convert a smart pointer-to-`T1` to a smart pointer-to-`T2` if and only if you can implicitly convert a `T1` to a `T2`.

This technique gives you exactly the behavior you want — almost. Suppose we augment our `MusicProduct` hierarchy with a new class, `CasSingle`, for representing cassette singles. The revised hierarchy looks like this:



Now consider this code:

```
template<class T>           // as before, including member tem-
class SmartPtr { ... };     // plate for conversion operators

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

void displayAndPlay(const SmartPtr<Cassette>& pc,
                    int howMany);

SmartPtr<CasSingle> dumbMusic(new CasSingle("Achy Breaky Heart"));

displayAndPlay(dumbMusic, 1);  // error!
```

In this example, `displayAndPlay` is overloaded, with one function taking a `SmartPtr<MusicProduct>` object and the other taking a `SmartPtr<Cassette>` object. When we invoke `displayAndPlay` with a `SmartPtr<CasSingle>`, we expect the `SmartPtr<Cassette>` function to be chosen, because `CasSingle` inherits directly from `Cassette` and only indirectly from `MusicProduct`. Certainly that's how it would work with dumb pointers. Alas, our smart pointers aren't that smart. They employ member functions as conversion operators, and as far as C++ compilers are concerned, all calls to conversion functions are equally good. As a result, the call to `displayAndPlay` is ambiguous, because the conversion from `SmartPtr<CasSingle>` to `SmartPtr<Cassette>` is no better than the conversion to `SmartPtr<MusicProduct>`.

Implementing smart pointer conversions through member templates has two additional drawbacks. First, support for member templates is rare, so this technique is currently anything but portable. In the future, that will change, but nobody knows just how far in the future that will be. Second, the mechanics of why this works are far from transparent, relying as they do on a detailed understanding of argument-matching rules for function calls, implicit type conversion functions, implicit instantiation of template functions, and the existence of member function templates. Pity the poor programmer who has never seen this trick before and is then asked to maintain or enhance code that relies on it. The technique is clever, that's for sure, but too much cleverness can be a dangerous thing.

Let's stop beating around the bush. What we really want to know is how we can make smart pointer classes behave just like dumb pointers for purposes of inheritance-based type conversions. The answer is simple: we can't. As Daniel Edelson has noted, smart pointers are smart, but they're not pointers. The best we can do is to use member tem-

plates to generate conversion functions, then use casts in those cases where ambiguity results. This isn't a perfect state of affairs, but it's pretty good, and having to cast away ambiguity in a few cases is a small price to pay for the sophisticated functionality smart pointers can provide.

Smart Pointers and const

Recall that for dumb pointers, `const` can refer to the thing pointed to, to the pointer itself, or both:

```
CD goodCD("Flood");

const CD *p;                    // p is a non-const pointer
                                // to a const CD object

CD * const p = &goodCD;         // p is a const pointer to
                                // a non-const CD object;
                                // because p is const, it
                                // must be initialized

const CD * const p = &goodCD;   // p is a const pointer to
                                // a const CD object
```

Naturally, we'd like to have the same flexibility with smart pointers. Unfortunately, there's only one place to put the `const`, and there it applies to the pointer, not to the object pointed to:

```
const SmartPtr<CD> p =          // p is a const smart ptr
  &goodCD;                      // to a non-const CD object
```

This seems simple enough to remedy — just create a smart pointer to a *`const`* `CD`:

```
SmartPtr<const CD> p =          // p is a non-const smart ptr
  &goodCD;                      // to a const CD object
```

Now we can create the four combinations of `const` and non-`const` objects and pointers we seek:

```
SmartPtr<CD> p;                 // non-const object,
                                // non-const pointer

SmartPtr<const CD> p;           // const object,
                                // non-const pointer

const SmartPtr<CD> p = &goodCD;// non-const object,
                                // const pointer

const SmartPtr<const CD> p = &goodCD;
                                // const object,
                                // const pointer
```

Like most C++ ointments, this one has a fly in it. Using dumb pointers, we can assign non-`const` pointers to `const` pointers and we can assign pointers to non-`const` objects to pointers to `const`s. For example:

```
CD *pCD = new CD("Famous Movie Themes");

const CD * pConstCD = pCD;                  // fine
```

But look what happens if we try the same thing with smart pointers:

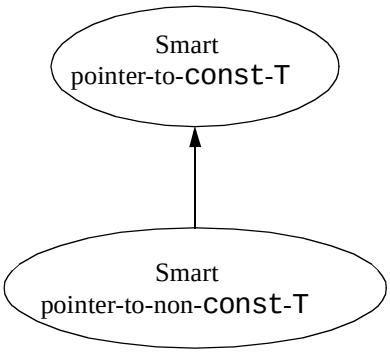
```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");

SmartPtr<const CD> pConstCD = pCD;          // fine?
```

`SmartPtr<CD>` and `SmartPtr<const CD>` are completely different types. As far as your compilers know, they are unrelated, so they have no reason to believe they are assignment-compatible. In what must be an old story by now, the only way these two types will be considered assignment-compatible is if you've provided a function to convert objects of type `SmartPtr<CD>` to objects of type `SmartPtr<const`

CD>. If you've got a compiler that supports member templates, you can use the technique shown above for automatically generating the implicit type conversion operators you need. (I remarked earlier that the technique worked anytime the corresponding conversion for dumb pointers would work, and I wasn't kidding. Conversions involving `const` are no exception.) If you don't have such a compiler, you have to jump through one more hoop.

Conversions involving `const` are a one-way street: it's safe to go from non-`const` to `const`, but it's not safe to go from `const` to non-`const`. Furthermore, anything you can do with a `const` pointer you can do with a non-`const` pointer, but with non-`const` pointers you can do other things, too (for example, assignment). Similarly, anything you can do with a pointer-to-`const` is legal for a pointer-to-non-`const`, but you can do some things (such as assignment) with pointers-to-non-`const`s that you can't do with pointers-to-`const`s.

These rules sound like the rules for public inheritance. You can convert from a derived class object to a base class object, but not vice versa, and you can do anything to a derived class object you can do to a base class object, but you can typically do additional things to a derived class object, as well. We can take advantage of this similarity when implementing smart pointers by having each smart pointer-to-`T` class publicly inherit from a corresponding smart pointer-to-`const`-`T` class:




```
template<class T>               // smart pointers to const
class SmartPtrToConst {         // objects

  ...                           // the usual smart pointer
                                // member functions

protected:
  union {
    const T* constPointee;      // for SmartPtrToConst access
    T* pointee;                 // for SmartPtr access
  };
};

template<class T>               // smart pointers to
class SmartPtr:                 // non-const objects
  public SmartPtrToConst<T> {
  ...                           // no data members
};
```

With this design, the smart pointer-to-non-`const`-`T` object needs to contain a dumb pointer-to-non-`const`-`T`, and the smart pointer-to-`const`-`T` needs to contain a dumb pointer-to-`const`-`T`. The naive way to handle this would be to put a dumb pointer-to-`const`-`T` in the base class and a dumb pointer-to-non-`const`-`T` in the derived class. That would be wasteful, however, because `SmartPtr` objects would contain two dumb pointers: the one they inherited from `SmartPtrToConst` and the one in `SmartPtr` itself.

This problem is resolved by employing that old battle axe of the C world, a union, which can be as useful in C++ as it is in C. The union is protected, so both classes have access to it, and it contains both of the necessary dumb pointer types. `SmartPtrToConst<T>` objects use the `constPointee` pointer, `SmartPtr<T>` objects use the `pointee` pointer. We therefore get the advantages of two different pointers without having to allocate space for more than one. Such is the beauty of a union. Of course, the member functions of the two classes must constrain themselves to using only the appropriate pointer, and you'll get no help from compilers in enforcing that constraint. Such is the risk of a union.

With this new design, we get the behavior we want:

```
SmartPtr<CD> pCD = new CD("Famous Movie Themes");
SmartPtrToConst<CD> pConstCD = pCD;        // fine
```

Evaluation of Smart Pointers

That wraps up the subject of smart pointers, but before we leave the topic, we should ask this question: are they worth the trouble, especially if your compilers lack support for member function templates?

Often they are. Reference-counting, for example, is greatly simplified by using smart pointers [cite 1 and 2]. Furthermore, some uses of smart pointers are sufficiently limited in scope that things like testing for nullness, conversion to dumb pointers, inheritance-based conversions, and support for pointers-to-`const`s are irrelevant. At the same time, smart pointers can be tricky to implement, understand, and maintain. Debugging code using smart pointers is more difficult than debugging code using dumb pointers. Try as you may, you will never succeed in designing a general-purpose smart pointer that can seamlessly replace its dumb pointer counterpart.

Smart pointers nevertheless make it possible to achieve effects in your code that would otherwise be difficult to implement. Smart pointers should be used judiciously, but every C++ programmer will find them useful at one time or another.

Acknowledgments

My discussion of smart pointers in this column and the two columns that preceded it is based in part on Steven Buroff's and Rob Murray's *C++ Oracle* column in the October 1993 *C++ Report*; on Daniel R. Edelson's classic paper, "Smart Pointers: They're Smart, but They're Not Pointers," in the proceedings of the 1992 USENIX C++ Conference; on section 15.9.1 of Bjarne Stroustrup's *The Design and Evolution of C++* (Addison-Wesley, 1994); on Gregory Colvin's "C++ Memory Management" class notes from C/C++ Solutions '95; and on Cay Horstmann's column in the March-April 1993 issue of the *C++ Report*. I developed some of the material myself, though. Really. Honest.

References

1. R. Murray, *C++ Strategies and Tactics*, Addison-Wesley, 1993, pp. 152-156.

2. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996, pp. 183-212.

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`.