Smart pointers, Part 1

Smart pointers are objects that are designed to look, act, and feel like built-in pointers, but to offer greater functionality. When you use smart pointers in place of C++'s built-in pointers (i.e., *dumb* pointers), you gain control over the following aspects of pointer behavior:

- **Construction and destruction**. You determine what happens when a smart pointer is created and destroyed. It is common to give smart pointers a default value of 0 to avoid the headaches associated with uninitialized pointers. Some smart pointers are made responsible for deleting the object they point to when the last smart pointer pointing to the object is destroyed. This can go a long way toward eliminating resource leaks.
- **Copying and assignment**. You control what happens when a smart pointer is copied or is involved in an assignment. For some smart pointer types, the desired behavior is to automatically copy or make an assignment to what is pointed to, i.e., to perform a deep copy. For others, only the pointer itself should be copied or assigned. For still others, these operations should not be allowed at all. Regardless of what behavior you consider "right," the use of smart pointers lets you call the shots.
- **Dereferencing**. What should happen when a client refers to the object pointed to by a smart pointer? You get to decide. You could, for example, use smart pointers to help implement a lazy fetching strategy whereby you don't fetch information about an object from a database until the pointer to that object is dereferenced in such a way that the information must be returned.

Smart pointers are generated from templates because, like built-in pointers, they must be strongly typed; the template parameter specifies the type of object pointed to. Most smart pointer templates look something like this:

```
template<class T>              // template for smart
class SmartPtr {               // pointer objects
public:
  SmartPtr(T* realPtr = 0);    // create a smart ptr to an
                               // obj given a dumb ptr to
                               // it; uninitialized ptrs
                               // default to 0 (null)

  SmartPtr(const SmartPtr& rhs); // copy a smart ptr

  ~SmartPtr();                 // destroy a smart ptr

  // make an assignment to a smart ptr
  SmartPtr& operator=(const SmartPtr& rhs);

  T* operator->() const;       // dereference a smart ptr
                               // to get at a member of
                               // what it points to
  T& operator*() const;         // defererence a smart ptr

private:
  T *pointee;                   // what the smart ptr
};                              // points to
```

The copy constructor and assignment operator are both shown public here. For smart pointer classes where copying and assignment are not allowed, they would typically be declared private. The two dereferencing operators are declared `const`, because dereferencing a pointer doesn't modify it (though it may lead to modification of what the pointer points to). Finally, each smart pointer-to-`T` object is implemented by containing a dumb pointer-to-`T` within it. It is this dumb pointer that does the actual pointing.

Before going into the details of smart pointer implementation, it's worth seeing how clients might use smart pointers. Consider a distributed system in which some objects are local and some are remote. Access to local objects is generally simpler and faster than access to remote objects, because remote access may require remote procedure calls or some other way of communicating with a distant machine.

For clients writing application code, the need to handle local and remote objects differently is a nuisance. It is more convenient to have all objects appear to be located in the same place. Smart pointers allow a library to offer this illusion:

```
template<class T>               // template for smart ptrs
class DBPtr {                   // to objects in a
public:                         // distributed DB

  DBPtr(T *realPtr = 0);        // create a smart ptr to a
                                // DB object given a local
                                // dumb pointer to it

  DBPtr(DataBaseID id);         // create a smart ptr to a
                                // DB object given its
                                // unique DB identifier

  ...                           // other smart ptr
};                              // functions as above

class Tuple {                   // class for database
public:                         // tuples
  ...
  void displayEditDialog();     // present a graphical
                                // dialog box allowing a
                                // user to edit the tuple

  bool isValid() const;         // return whether *this
};                              // passes validity check

// template class for making log entries whenever a T
// object is modified; see below for details
template<class T>
class LogEntry {
public:
  LogEntry(const T& objectToBeModified);
  ~LogEntry();
};

// note that the parameter pt below is a smart ptr object
void editTuple(DBPtr<Tuple>& pt)
{
  LogEntry<Tuple> entry(*pt);   // make log entry for this
                                // editing operation; see
                                // below for details

  // repeatedly display edit dialog until valid values
  // are provided
  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

The tuple to be edited inside `editTuple` may be physically located on a remote machine, but the programmer writing `editTuple` need not be concerned with such matters; the smart pointer class hides that aspect of the system. As far as the programmer is concerned, all tuples are accessed through objects that, except for how they're declared, act just like run-of-the-mill built-in pointers.

Notice the use of a `LogEntry` object in `editTuple`. A more conventional design would have been to surround the call to `displayEditDialog` with calls to begin and end the log entry. In the approach shown here, the `LogEntry`'s constructor begins the log entry and its destructor ends the log entry. Using an object to begin and end logging is more robust in the face of exceptions than explicitly calling functions [cite 1, 2, 3], so you should accustom yourself to using classes like `LogEntry`. Besides, it's easier to create a single `LogEntry` object than to add separate calls to start and stop an entry.

As you can see, using a smart pointer isn't much different from using the dumb pointer it replaces. That's testimony to the effectiveness of encapsulation. Clients of smart pointers are *supposed* to be able to treat them as dumb pointers. As we shall see, sometimes the substitution is more transparent than others.

Construction, Assignment, and Destruction of Smart Pointers

Construction of a smart pointer is usually straightforward: locate an object to point to (typically by using the smart pointer's constructor arguments), then make the smart pointer's internal dumb pointer point there. If no object can be located, set the internal pointer to 0 or signal an error (possibly by throwing an exception).

Implementing a smart pointer's copy constructor, assignment operator(s) and destructor is complicated somewhat by the issue of *ownership*. If a smart pointer *owns* the object it points to, it is responsible for deleting that object when it (the smart pointer) is destroyed. This assumes the object pointed to by the smart pointer is dynamically allocated. Such an assumption is common when working with smart pointers.

Consider the `auto_ptr` template from the standard C++ library. An `auto_ptr` object is a smart pointer that points to a heap-based object until it (the `auto_ptr`) is destroyed. When that happens, the `auto_ptr`'s destructor deletes the pointed-to object. The essence of the `auto_ptr` template might be implemented like this:

```
template<class T>
class auto_ptr {
public:
  auto_ptr(T *ptr = 0): pointee(ptr) {}
  ~auto_ptr() { delete pointee; }
  ...

private:
  T *pointee;
};
```

This works fine provided only one `auto_ptr` owns an object. But what should happen when an `auto_ptr` is copied or assigned?

```
auto_ptr<TreeNode> ptn1(new TreeNode);

auto_ptr<TreeNode> ptn2 = ptn1;    // call to copy ctor;
                                   // what should happen?

auto_ptr<TreeNode> ptn3;

ptn3 = ptn2;                       // call to operator=;
                                   // what should happen?
```

If we just copied the internal dumb pointer, we'd end up with two `auto_ptr`s pointing to the same object. This would lead to grief, because each `auto_ptr` would delete what it pointed to when the `auto_ptr` was destroyed. That would mean we'd delete an object more than once. The results of such double-deletes are undefined (and are frequently disastrous).

An alternative would be to create a new copy of what was pointed to by calling `new`. That would guarantee we didn't have too many `auto_ptr`s pointing to a single object, but it might engender an unacceptable performance hit for the creation (and later destruction) of the new object. Furthermore, we wouldn't necessarily know what type of object to create, because an `auto_ptr<T>` object need not point to an object of type `T`; it might point to an object of a type *derived* from `T`. Virtual constructors [cite 6, 7] can help solve this problem, but it seems inappropriate to require their use in a lightweight, general-purpose class like `auto_ptr`.

The problems would vanish if `auto_ptr` prohibited copying and assignment, but a more flexible solution was adopted for the `auto_ptr` classes: object ownership is *transferred* when an `auto_ptr` is copied or assigned:

```
template<class T>
class auto_ptr {
public:
  ...

  auto_ptr(auto_ptr<T>& rhs);   // copy constructor

  auto_ptr<T>&                  // assignment
  operator=(auto_ptr<T>& rhs);  // operator

  ...
};

template<class T>
auto_ptr<T>::auto_ptr(auto_ptr<T>& rhs)
{
  pointee = rhs.pointee;        // transfer ownership of
                                // *pointee to *this

  rhs.pointee = 0;              // rhs no longer owns
}                               // anything

template<class T>
auto_ptr<T>& auto_ptr<T>::operator=(auto_ptr<T>& rhs)
{
  if (this == &rhs)             // do nothing if this
    return *this;               // object is being assigned
                                // to itself

  delete pointee;               // delete currently owned
                                // object

  pointee = rhs.pointee;        // transfer ownership of
  rhs.pointee = 0;              // *pointee from rhs to *this

  return *this;
}
```

Notice that the assignment operator must delete the object it owns before assuming ownership of a new object. If it failed to do this, the object would never be deleted. Remember, nobody but the `auto_ptr` object owns the object the `auto_ptr` points to.

Because object ownership is transferred when `auto_ptr`'s copy constructor is called, passing `auto_ptr`s by value is often a *very* bad idea. Here's why:

```
// this function will often lead to disaster
void printTreeNode(ostream& s, auto_ptr<TreeNode> p)
{ s << *p; }
```

```
int main()
{
  auto_ptr<TreeNode> ptn(new TreeNode);
  ...
  printTreeNode(cout, ptn);     // pass auto_ptr by value
  ...
}
```

When printTreeNode's parameter p is initialized (by calling auto_ptr's copy constructor), ownership of the object pointed to by ptn is transferred to p. When printTreeNode finishes executing, p goes out of scope and its destructor deletes what it points to (which is what ptn used to point to). ptn, however, no longer points to anything (its underlying dumb pointer is null), so just about any attempt to use it after the call to printTreeNode will yield undefined behavior. Passing auto_ptrs by value, then, is something to be done only if you're *sure* you want to transfer ownership of an object to a (transient) function parameter. Only rarely will you want to do this.

This doesn't mean you can't pass auto_ptrs as parameters, it just means that pass-by-value is not the way to do it. Pass-by-reference-to-const is:

```
// this function behaves much more intuitively
void printTreeNode( ostream& s,
                    const auto_ptr<TreeNode>& p)
{ s << *p; }
```

In this function, p is a reference, not an object, so no constructor is called to initialize p. When ptn is passed to this version of printTreeNode, it retains ownership of the object it points to, and ptn can safely be used after the call to printTreeNode. Thus, passing auto_ptrs by reference-to-const avoids the hazards arising from pass-by-value.

The notion of transferring ownership from one smart pointer to another during copying and assignment is interesting, but you may have been at least as interested in the unconventional declarations of the copy constructor and assignment operator. These functions normally take const parameters, but above they do not. In fact, the code above *changes* these parameters during the copy or the assignment. In other words, auto_ptr objects are modified if they are copied or are the source of an assignment!

Yes, that's exactly what's happening. Isn't it nice that C++ is flexible enough to let you do this? If the language required that copy constructors and assignment operators take const parameters, you'd probably have to cast away the parameters' constness or play other games to implement ownership transferral. Instead, you get to say exactly what you want to say: when an object is copied or is the source of an assignment, that object is changed. This may not seem intuitive, but it's simple, direct, and, in this case, accurate.

If you find this examination of auto_ptr member functions interesting, you may wish to see a complete implementation. You'll find one in *More Effective C++* [cite 4], where you'll also see that the auto_ptr template in the standard C++ library has copy constructors and assignment operators that are more flexible than those described here. In the standard auto_ptr template, those functions are member function *templates* [cite 5], not just member functions. (I'll describe member function templates in a later column.)

A smart pointer's destructor often looks like this:

```
template<class T>
SmartPtr<T>::~SmartPtr()
{
  if (*this owns *pointee) {
    delete pointee;
  }
}
```

Sometimes there is no need for the test. An `auto_ptr` always owns what it points to, for example. At other times the test is a bit more complicated. A smart pointer that employs reference counting must adjust a reference count before determining whether it has the right to delete what it points to.

Implementing the Dereferencing Operators

Let us now turn our attention to the very heart of smart pointers, the `operator*` and `operator->` functions. The former returns the object pointed to. Conceptually, this is simple:

```
template<class T>
T& SmartPtr<T>::operator*() const
{
  perform "smart pointer" processing;

  return *pointee;
}
```

First the function does whatever processing is needed to initialize or otherwise make `pointee` valid. For example, if lazy fetching is being used, the function may have to conjure up a new object for `pointee` to point to. Once `pointee` is valid, the `operator*` function just returns a reference to the pointed-to object.

Note that the return type is a *reference*. It would be disastrous to return an *object* instead, though compilers will let you do it. Bear in mind that `pointee` need not point to an object of type `T`; it may point to an object of a class *derived* from `T`. If that is the case and your `operator*` function returns a `T` object instead of a reference to the actual derived class object, your function will return an object of the wrong type! Virtual functions invoked on the object returned from your star-crossed `operator*` will not invoke the function corresponding to the dynamic type of the pointed-to object. In essence, your smart pointer will not properly support virtual functions, and how smart is a pointer like that? Besides, returning a reference is more efficient anyway, because there is no need to construct a temporary object. This is one of those happy occasions when correctness and efficiency go hand in hand.

If you're the kind who likes to worry, you may wonder what you should do if somebody invokes `operator*` on a null smart pointer, i.e., one whose embedded dumb pointer is null. Relax. You can do anything you want. The result of dereferencing a null pointer is undefined, so there is no "wrong" behavior. Wanna throw an exception? Go ahead, throw it. Wanna call `abort` (possibly by having an `assert` call fail)? Fine, call it. Wanna walk through memory setting every byte to your birth date modulo 256? That's okay, too. It's not nice, but as far as the language is concerned, you are completely unfettered.

The story with `operator->` is similar to that for `operator*`, but before examining `operator->`, let us remind ourselves of the unusual meaning of a call to this function. Consider again the `editTuple` function that uses a smart pointer-to-`Tuple` object:

```
void editTuple(DBPtr<Tuple> pt)
{
  LogEntry<Tuple> entry(*pt);

  do {
    pt->displayEditDialog();
  } while (pt->isValid() == false);
}
```

The statement

```
pt->displayEditDialog();
```

is interpreted by compilers as:

```
(pt.operator->())->displayEditDialog();
```

That means that whatever `operator->` returns, it must be legal to apply the member-selection operator (`->`) to it. There are thus only two things `operator->` can return: a dumb pointer to an object or another smart pointer object. Most of the time, you'll want to return an ordinary dumb pointer. In those cases, you implement `operator->` as follows:

```
template<class T>
T* SmartPtr<T>::operator->() const
{
  perform "smart pointer" processing;

  return pointee;
}
```

This will work fine. Because this function returns a pointer, virtual function calls via `operator->` will behave the way they're supposed to.

For many applications, this is all you need to know about smart pointers. Most reference-counting implementations, for example, draw on no more functionality than we've discussed here. If you want to push your smart pointers further, however, you must know more about dumb pointer behavior and how smart pointers can and cannot emulate it. If your motto is "Most people stop at the Z — but not me!", the material in my next two columns is for you.

References

1. M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990, pp. 358-359.

2. B. Stroustrup, *The C++ Programming Language*, 2nd Ed., Addison-Wesley, 1991, pp. 308-314.

3. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996, pp. 45-58.

4. *Ibid*, pp. 291-294.

5. J. Lajoie, "More on Templates," *C++ Report*, Nov.-Dec. 1994, pp. 48-49.

6. J. Copeline, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992, pp. 140-148.

7. S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996, pp. 123-130.

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`.