C/C++ Contributing Editors

C++ Made Easier: The Rule of Three

Andrew Koenig and Barbara E. Moo

What you leave out of a class can cause you just as much trouble as what you put in. This simple rule will save you the most common mistakes.

Introduction

It seems that every week or two, someone posts a programming example to the comp.lang.c++ newsgroup on Usenet that involves a program that fails because of a class that is missing a copy constructor or an assignment operator. Invariably, the trouble with the class in question is obvious to anyone who understands the Rule of Three.

The Rule of Three [1] says that there are three member functions that go together: the destructor, the copy constructor, and the assignment operator. A class that defines a destructor should almost always define the other two members. Moreover, a class that defines a copy constructor or assignment operator should usually define the other two members as well [2].

This article explains the reasoning behind the Rule of Three.

An Example and a Misconception

Let's take a look at a class that someone might write as part of an effort to understand how the standard-library **vector** class works. Indeed, before implementations of the standard library were widely available, we used to use a class much like this one in the summer continuing-education course that we taught at Stanford:

```
// This class contains a subtle error
class IntVec {
public:
   IntVec(int n): data(new int[n]) { }
   ~IntVec() { delete[] data; };
   int& operator[](int n)
      { return data[n]; }
   const int& operator[](int n) const
      { return data[n]; }

private:
   int* data;
};
```

This class attempts to simplify the **vector** class by dealing only with arrays of integers, and by not worrying about the ability of **vector**s to grow and shrink on command. Constructing an **IntVec** object allocates enough memory for the given number of integers, destroying the object deallocates the memory, and indexing it accesses the given element.

The trouble with this example is that it has simplified matters too much — to the extent that copying an object of this class will cause serious trouble:

```
int main()
{
   IntVec x(100);
   IntVec y = x;   // Trouble!
   return 0;
}
```

The reason for the trouble is that the **IntVec** class does not explicitly define a copy constructor. When a class does not define a copy constructor, the implementation synthesizes one, defining copying an object of the class in terms of copying the members of the class. In other words, when we initialize **y** as a copy of **x**, the implementation handles that definition by initializing **y.data** to be a copy of **x.data**. Forming this copy is not harmful by itself. However, when the program terminates, the local variables **x** and **y** will both be destroyed, which will result in executing **delete[]** on **x.data** and **y.data**. Because **x.data** and **y.data** have the same value, these two **delete[]** operations will try to free the same memory twice, the effect of which is undefined.

We have seen the claim that the sign of trouble in the **IntVec** class is that it contains a pointer, and that classes that have pointers as data members should have copy constructors. We disagree with this claim, having come to realize that the pointer by itself is harmless. After all, a pointer is just a value. Why should there be anything intrinsically more harmful about copying a pointer than about copying any other value?

The true sign of trouble in this class is that the class has a destructor. In almost all cases, when a class has a destructor, it also needs a copy constructor and an assignment operator. To see that it is the destructor that is fundamental, try removing it. Doing so creates a memory leak, of course, but it also avoids the trouble associated with multiple deletion. If you are willing to tolerate the memory leak that stems from the lack of a destructor, there is no additional reason why this class needs a copy constructor or assignment operator.

Fixing Our Class

Having written a class that requires a copy constructor and an assignment operator, how shall we define those members? If we don’t know what else to do, we can punt:

```
// This class corrects the error
// by brute force
class IntVec {
public:
   IntVec(int n): data(new int[n]) { }
   ~IntVec() { delete[] data; };
   int& operator[](int n)
      { return data[n]; }
   const int& operator[](int n) const
      { return data[n]; }
```

```
private:
   int* data;

   // these two member functions added
   IntVec(const IntVec&);
   IntVec& operator=(const IntVec&);
};
```

By declaring a copy constructor and assignment operator explicitly, we inhibit the compiler's built-in versions of those functions. By making the copy constructor and assignment operator **private**, we prevent anyone from using them. Because no one can use them, we don't have to define them: if a member function is not **virtual**, then it needs to be defined only if someone uses it.

This solution to the problem has the great virtue of requiring almost no thought. As long as no one tries to copy an **IntVec**, we don't need to worry about how to do so. Moreover, any attempts to copy an **IntVec** will be caught during compilation.

Of course, we may wish to make our class more useful, rather than merely outlawing operations that we were unwilling to take the trouble to define. To do so, we must think about what copying an **IntVec** object means. There are several possible meanings, the most straightforward of which is probably to say that copying an **IntVec** object copies its contents. In that case, we can define the copy constructor and assignment operator this way:

```
// This class corrects the error by
// defining copying and assignment
class IntVec {
public:
   IntVec(int n): data(new int[n]), size(n) { }
   ~IntVec() { delete[] data; };
   int& operator[](int n)
      { return data[n]; }
   const int& operator[](int n) const
      { return data[n]; }

   IntVec(const IntVec& v):
      data(new int[v.size]),
      size(v.size) {
      std::copy(data, data + size, v.data);
   }
   IntVec&
   operator=(const IntVec& v) {
      int* newdata = new int[v.size];
      std::copy(v.data,v.data+v.size, newdata);
      delete[] data;
      data = newdata;
      size = v.size;
      return *this;
   }

private:
   int* data;
   int size;
};
```

You can see that our modifications to this little class have more than doubled its size. These modifications all stem from our decision to define the copy constructor and assignment operator in ways that are consistent with the destructor.

The Reasons for the Requirements

In order to see why destructors require copy constructors and assignment operators, think about what a destructor does: it contains code that is supposed to run whenever an object of that class is destroyed. What does this code do? It should be clear that if the destructor were to affect only the contents of the object itself, the destructor would be useless — if an object is in the process of being destroyed, any change to that object will soon be obliterated. Therefore, for a destructor to be useful, it must affect a part of the program's state other than the object itself.

Whatever this effect might be, it must be important. Otherwise, why bother with a destructor at all? For example, in our **IntVec** class, the effect is to deallocate the memory that the constructor allocated.

We can conclude, then, that if our class has a destructor, that destructor does something important to the state of the program outside the class object itself. Now, let's suppose that our class does not have a copy constructor. Then copying an object of that class will copy all of its data members. When these two objects are destroyed, the destructor will run twice. Moreover, the information available to the destructor in the object being destroyed will be the same in each case, because each data member of the copy will be a copy of the corresponding data member of the original. We already know that we care whether the destructor was executed once or not at all. It is therefore highly likely that we also care whether the destructor is executed twice, as opposed to once. Yet if we do not have a copy constructor, copying an object will cause a destructor that would otherwise have been executed once to be executed twice under the same conditions. Such duplicate executions are a recipe for trouble.

A similar argument lets us conclude that a class with a destructor must also have an assignment operator. If such a class does not have an explicitly defined assignment operator, assigning one object of that class to another will assign all the source's data members to the corresponding data members of the destination. After that assignment, the destination's data members will be copies of the source's data members, and the destructor will be in exactly the same kind of trouble that it would have been in had the object been copied rather than assigned.

Although a class with a destructor almost always needs a copy constructor and an assignment operator, the reverse is not always true. The reason is that not every copy constructor or assignment operator allocates resources, so not every copy constructor or assignment operator requires a destructor in order to free those resources. For example, a copy constructor might exist because it does *less* work than the default copy constructor would do. As an example of such a copy constructor, consider a class each instance of which contains data (of type **Data**) and a cache (of type **Cache**):

```
class Thing {
public:
   Thing() { /* ... */ }
   Thing(const Thing& t):
```

```
      data(t.data)
   { }  // don't copy the cache
   Thing& operator=(const Thing& t) {
      data = t.data; // copy the data

      // clear the cache
      cache = Cache();
   }
private:
   Data data;
   Cache cache;
};
```

Here, the copy constructor exists in order to *prevent* the cache from being copied, which the default copy constructor would do. Similarly, the assignment operator explicitly clears the cache rather than assigning it from **t.cache**. It should not be hard to imagine that such a class might legitimately be able to do without an explicit destructor.

However, if a copy constructor does anything that is not ephemeral, the class will need a destructor. For example, a class with a copy constructor that allocates a resource will need a destructor that deallocates that resource, unless the author is willing for the class to leave the resources allocated.

Another example of a class that requires a copy constructor and assignment operator is one in which one part of the class object refers to another part. For example, consider a class that represents a string with a marker somewhere in it:

```
class String_with_marker {
   // ...
private:
   std::string str;
   int marker;
};
```

Here, the **marker** member represents the index of the character at which the marker is located. From what we've shown here about this class, there is no need for a copy constructor or assignment operator. However, suppose that instead of using an integer, we want to use an iterator to mark the position:

```
class String_with_marker {
   // ...
private:
   std::string str;
   // changed
   std::string::iterator marker;
};
```

Now, although we still do not need a destructor, we absolutely do need a copy constructor and an assignment operator. Otherwise, copying an object of this class will copy both the string and the corresponding iterator, and copying the iterator will yield an iterator that still refers to an element of the original string. We might define these members as follows:

```
class String_with_marker {
   // ...
public:
```

```
   // added
   String_with_marker(const String_with_marker& s):
      str(s.str),
      marker(str + (s.marker - s.str.begin())) { }
   String_with_marker& operator=(const String_with_marker& s)
   {
      str = s.str;
      marker = str + (s.marker - s.str);
      return *this;
   }

private:
   std::string str;
   std::string::iterator marker;
};
```

Conclusion

The Rule of Three is really two rules:

- If a class has a nonempty destructor, it almost always needs a copy constructor and an assignment operator.
- If a class has a nontrivial copy constructor or assignment operator, it usually needs both of these members and a destructor as well.

A destructor usually deallocates a resource. If the resource needs explicit deallocation, it is a good bet that the implementation-generated copy constructor and assignment operator will not allocate another instance of that resource properly. Therefore, whenever a class has a nontrivial destructor, you should assume that a copy constructor and assignment operator are required unless you can prove otherwise. Similarly, if you have a copy constructor or assignment operator, you probably need both of them and a destructor too.

Some resources cannot easily be copied. For example, imagine a class that is designed to ensure exclusive access to a device. Such a class would set a lock in its constructor and clear it in its destructor. How would it make sense to copy such a class? Its whole purpose is to ensure exclusivity.

Designing a copy constructor for a class might be more trouble than it is worth. Consider our **IntVec** class, where adding a copy constructor and assignment operator more than doubled the size of the class.

If you need a copy constructor or assignment operator, and you can’t find a reasonable way to implement them or don’t want to do so, consider making them private so that programmers won’t try to use them by accident. In all other cases, you should write an assignment operator and copy constructor whenever you write a destructor.

Notes

[1] Marshall Cline coined the term in 1991.

[2] Classes often have an empty virtual destructor, which is there only because it is virtual and not to do any actual work. Such destructors don’t count for the purpose of this

discussion.

Andrew Koenig *is a member of the Large-Scale Programming Research Department at AT&T's Shannon Laboratory, and the Project Editor of the C++ standards committee. A programmer for more than 30 years, 15 of them in C++, he has published more than 150 articles about C++ and speaks on the topic worldwide. He is the author of* C Traps and Pitfalls *and co-author of* Ruminations on C++.

Barbara E. Moo is an independent consultant with 20 years' experience in the software field. During her nearly 15 years at AT&T, she worked on one of the first commercial projects ever written in C++, managed the company's first C++ compiler project, and directed the development of AT&T's award-winning WorldNet Internet service business. She is co-author of Ruminations on C++ *and lectures worldwide.*