Lecture on Assignment Operator

Topics:

1. Operator Overloading – Article

We acknowledge fully the contribution of the author of the article. Using the article is only to present information to the readers and by/with no other means!
Tuan T. Nguyen

The original article can be found at
http://icu.sourceforge.net/docs/papers/cpp_report/the_anatomy_of_the_assignment_operator.html

The Anatomy of the Assignment Operator

by Richard Gillam
Senior Software Engineer, Text & International
Taligent, Inc.

My team recently hired someone. Normally, this wouldn't be such a big deal, but we've been looking for someone for a year and a half. In this time, we've interviewed at least a dozen candidates and phone-screened at least a couple dozen more. Practically every candidate we've talked to had at least two years of C++ experience, rated himself a 7 or 8 on a scale of 10 in C++ skill, and had one or two lucrative offers on the table. Unfortunately, we would have rated almost all of them somewhere between a 4 and a 6. In my opinion, this goes to show you that working with C++ for a long time doesn't guarantee you really understand the language.

Over this time, I've developed a stock interview question that's proven to be a pretty good gauge of C++ knowledge. No one has yet been able to just rip out the correct answer, but we've had several, including the guy we hired, who understood the important issues and were able to get the question right with prompting. As a public service, I'd like to share my stock question and its answer with you and explore the various programming issues it presents.

The question is as follows:

Consider the following class definition:

```
class TFoo : public TSuperFoo {
    TBar* fBar1;
    TBar* fBar2;
    // various method definitions go here...
}
```

You have a class, `TFoo`, which descends from a class, `TSuperFoo`, and which has two data members, both of which are pointers to objects of class `TBar`. For the purposes of this exercise, consider both pointers to have owning semantics and `TBar` to be a monomorphic class. Write the assignment operator for this class.

This seems like a simple enough exercise, but it gets at some interesting issues. It's a good way to test a programmer's grasp of C++ syntax and C++ style, but more importantly, it tests the programmer's knowledge of C++ memory management and exception handling.

For the impatient among you, let's cut right to the chase: One correct answer to this question would look something like this:[*]

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        TBar* bar1 = 0;
        TBar* bar2 = 0;

        try {
            bar1 = new TBar(*that.fBar1);
            bar2 = new TBar(*that.fBar2);
        }
        catch (...) {
            delete bar1;
            delete bar2;
            throw;
        }

        TSuperFoo::operator=(that);
        delete fBar1;
        fBar1 = bar1;
        delete fBar2;
        fBar2 = bar2;
    }
    return *this;
}
```

Yes, it's a lot of code. Yes, it's ugly. But all the code you see here is necessary. We'll go through it all piece by piece and see why this is.

"But I never have to write an assignment operator!"

The first reaction I usually get from people is something along the lines of "But I never have to write assignment operators." You should. If you've ever created a new class, you've needed to write an assignment operator.

Let's examine why this is so. In C++, there are three things every object is expected to be able to do: An object should be able to initialize itself to a default state, it should be able to initialize itself from another instance of the same class, and it should be able to assume the semantic state of another instance of the same class. In C++, these operations are expressed with the default constructor (e.g., `TFoo::TFoo()`), the copy constructor (`TFoo::TFoo(const TFoo&)`), and the assignment operator (`TFoo::operator=(const TFoo&)`).

These three functions are special in C++: If you don't provide them yourself, C++ provides them for you. And automatically makes them public. Among other things, this means you have to define these operations even if you *don't want* a client to be able to copy or default-construct a particular class. If you don't want a class to be copied, for example, you have to define an empty copy constructor and assignment operator yourself and make them private or protected.

Furthermore, the compiler isn't guaranteed to create versions of these classes that do exactly what you want them to do. For copying and assignment, for example, the automatically-generated code will do a *shallow memberwise copy*. If your class has pointer members, this is practically never what you want, and even when you don't have pointer members, this isn't always the right behavior. It's definitely not what we want in our example.

Even when the default versions of the special functions do what you want them to, it's still generally a good policy to always spell that out explicitly by writing them yourself. It avoids ambiguity, and it forces you to think more about what's going on inside your class. Always give any new class a default constructor, a copy constructor, and an assignment operator.

Copy vs. assign

Another misconception I see often is a fuzzy idea of the difference between the copy constructor and the assignment operator. They're not the same thing, although they're similar. Let's take a moment to look at the difference.

The copy constructor and assignment operator do similar things. They both copy state from one object to another, leaving them with equivalent semantic state. In other words, both objects will behave the same way and return the same results when their methods are called. If they have public data members (generally a bad idea), they have the same values. This doesn't necessarily mean that the objects are identical: some purely internal data members (such as caches) might not be copied, or data members pointing to other objects might end up pointing to different objects that are themselves semantically equivalent, rather than pointing to the same objects.

The difference between the copy constructor and assignment operator is that the copy constructor is a *constructor* — a function whose job it is to turn raw storage into an object of a specific class. An assignment operator, on the other hand, copies state between two *existing objects.* In other words, an assignment operator has to take into account the current state of the object when copying the other object's state into it. The copy constructor is creating a new object from raw storage and knows it's writing over garbage. For many classes, the current state of the object doesn't matter and both functions do the same thing. But for some classes (including the one in our example), the current state does matter, and the assignment operator is more complicated.

Defining the Function

What parameters does the function take? C++ requires that an assignment operator take one parameter: the thing on the right-hand side of the = sign. This can be of any type, but the assignment operator that C++ automatically generates for you (and therefore, the one we're interested in here) is the one where you have the same type of object on both sides of the = sign. That means the parameter is either an instance of or a reference to an instance of the same class as the object on the left-hand side. You'll pretty much always want to use a reference rather than a full-blown instance of the class (i.e., pass by reference instead of pass by value). This is because passing an object by value requires creating a new instance of the class with the same state as the object passed as a parameter: in other words, its copy constructor must be called. This isn't necessary, and it wastes time. The parameter can be either a const or a non-const reference, but since it would be terrible form for the assignment operator to have side effects on the object on the right-hand side, you should use a const reference.

What does the function return? An assignment operator can return anything it wants, but the standard C and C++ assignment operators return a reference to the left-hand operand. This allows you to chain assignments together like so:

```
x = y = z = 3;
```

Unless you have a *really good reason,* you want to follow this convention. Returning a reference to the right-hand operand or a value (i.e., another whole `TFoo`) would both still allow the simple chain described above to work, but have subtle differences in semantics from the way the standard operators do it that would come out in more complicated expressions involving assignment. Returning a value also forces unnecessary trips through the object's copy constructor, costing you in performance.

So the outer shell of a properly-written assignment operator would look like this:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    // copy the state...
    return *this;
}
```

(By the way, note that the return statement is "`return *this`", rather than "`return this`". That's because `this` is a *pointer* to `TFoo`. You have to dereference it to get a reference to `TFoo`. Of course, if you forget this, the compiler will remind you.)

Virtual or non-virtual? I had one applicant suggest that `operator=` should be a virtual function. Let's take a look at this issue. Many C++ programmers are trained to make everything virtual, and in fact, some older frameworks do just that. In the specific example of the assignment operator, however, it's not a good idea. An override of a virtual function has to take the same parameters as the function it's overriding. Therefore, `TFoo`'s `operator=` function would have to be declared as

```
virtual TSuperFoo& TFoo::operator=(TSuperFoo& that);
```

You could declare the function this way and still have it return `this`, of course, because a reference to a `TFoo` *is* a reference to a `TSuperFoo`, but we have no way of knowing whether `that` is a reference to a `TFoo` or a reference to some other subclass of `TSuperFoo`. If it's not a `TFoo`, you have several problems. You'd have to check `that`'s class, which can be expensive. If it isn't a `TFoo`, you obviously wouldn't want to try to carry out the assignment, but then you'd have to define some kind of error-handling protocol to handle this situation. Better just to make `operator=` take the right type and let the compiler check the classes of your operands for you at compile time.

Of course, as soon as each class has operands with different types, the functions have different signatures and the `operator=` function is no longer being overridden. So it doesn't make sense to make `operator=` virtual.

Owning and Aliasing Pointers

Okay, now that we've got the preliminaries out of the way, we can get into the actual nitty-gritty of having our assignment operator actually perform an assignment. Let's refresh our memory of what the object we're working on looks like:

```
class TFoo : public TSuperFoo {
    TBar* fBar1;
    TBar* fBar2;
    // method definitions...
};
```

It seems the obvious way to do the assignment would be this:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    fBar1 = that.fBar1;
    fBar2 = that.fBar2;
    return *this;
}
```

Unfortunately, that's the wrong answer here. Remember that in the original question I said that `fBar1` and `fBar2` are *owning* pointers. To understand why the above example won't do what we want, we need to take a look at the unique problems of C++ memory management.

Because of its evolution from C, C++ is much closer to the hardware than most other object-oriented languages. One of the chief consequences of this is that you have to do your own memory management. Every `new` that happens during a program's execution must be balanced by one and only one `delete`. You don't want objects you've allocated to clutter up memory after you're done with them, you don't want to try to delete an object more than once and you don't want to access an object after you've deleted it. Double-deleting an object can corrupt the memory manager's free list, leading to crashes down the road; reading through a pointer to a deleted object (a "dangling pointer") can lead to wrong results; and writing through a dangling pointer can corrupt other objects or cause crashes. Failing to delete an object you're done with (a "memory leak") is less obviously malignant, but can seriously degrade performance and eventually cause crashes when the system runs out of memory.

In a system of any complexity, sticking to this "one delete for every new" rule can be quite difficult, so a strict protocol for managing memory is necessary. The basic rule we follow at Taligent is that for every object in the runtime environment, there is one and only one pointer to it through which the object can be deleted. This pointer is an "owning pointer," and the object or function containing that pointer is the object's "owner." All other

pointers to the object are called "aliasing pointers." The owner of the object expects to delete the object; objects with aliasing pointers don't.

So when we say that `TFoo`'s two `TBar` pointers are owning pointers, we're saying that `TFoo` expects to delete those `TBars`. In other words, its destructor looks like this:

```
TFoo::~TFoo()
{
    delete fBar1;
    delete fBar2;
}
```

You can see that if more than one `TFoo` object points to a given `TBar`, then as soon as one of those `TFoos` is deleted (taking the `TBars` down with it), the other `TFoos` are hosed. The next time any one of them tried to access one of its `TBar` objects, it'd be reading or writing through a dangling pointer, with potentially disastrous consequences. Therefore, every `TFoo` must have its own unique `TBar` objects, which means our assignment operator must create new copies of the source object's `TBars` for the destination object to point to.

In some cases, of course, it's overkill to make a copy of an object, because the current owner of the object is just going to delete that object after passing its content on to another object. In other words, one object is *transferring ownership* of an object to another object. This happens quite frequently, in fact. A simple factory method starts out with ownership of the object it creates, but when it returns its value, it passes ownership of that object to the caller. Its return value is an owning pointer. At other times, a function returns a pointer to an object but intends that the caller merely use it for a short time to perform some operation. Ownership is not transferred; the return value is an aliasing pointer.

When you have functions that return pointers or have pointer parameters, you must make it explicit whether the function transfers ownership, and you must then make sure that code calling the function upholds these semantics. C++ doesn't do any of this for you. Sometimes you can do this through the parameter types (references are virtually always aliases, and const pointers are always aliases), and sometimes you have to do it with naming conventions (at Taligent, for example, we use "adopt," "orphan," and "create" in the names of functions that transfer ownership).

In the case of the assignment operator, our parameter is a const reference to another `TFoo`. That alone signifies that we are not taking ownership of its internal state (there are some very rare, but important, exceptions to this rule—we'll look at one later). Since `TFoo`'s pointers are defined as owning pointers, however, we have to reconcile the difference in semantics by making new copies of the objects the other `TFoo` points to.

As an aside, it's worth mentioning that there are occasions where copying objects to maintain ownership semantics is too expensive. In these situations, a technique called *reference counting* can be used. We'll touch briefly on reference counting later.

memcpy() is evil!

So we have to copy the `TBars` in our assignment operator. I've seen a lot of interesting attempts to do this. The most frightening was

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    memcpy(&fBar1, &that.fBar1, sizeof(fBar1));
    memcpy(&fBar2, &that.fBar2, sizeof(fBar2));

    return *this;
}
```

I really hope this guy was just nervous. This code just copies the *pointer values* from one `TFoo` to the other. In other words, it's an ugly way of doing

```
fBar1 = that.fBar1;
fBar2 = that.fBar2;
```

Closer to the mark, but ever scarier in terms of its results, is

```
memcpy(fBar1, that.fBar1, sizeof(TBar));
memcpy(fBar2, that.fBar2, sizeof(TBar));
```

This would copy the data members of `that`'s `TBars` into `this`'s `TBars`, so `this` and `that` retain their own separate `TBar` objects. So we're doing well so far. The problem is that it bypasses the assignment operator and copy constructor for `TBar`, so if `TBar` has any owning pointers of its own, you have the same problem. You'll also have problems if `TBar` owns locks or system resources that need to be properly cleaned up or duplicated when you change the internal state of the object. And, of course, if any of these pointers is `NULL`, you'll probably crash.

Finally, I had one applicant propose

```
fBar1 = new TBar;
memcpy(fBar1, that.fBar1, sizeof(TBar));
fBar2 = new TBar;
memcpy(fBar2, that.fBar2, sizeof(TBar));
```

This is kind of a cheap way of initializing brand-new `TBars` from existing ones, or copy constructing them without using the copy constructor. It suffers from all of the same limitations as the previous example, plus an additional one we'll get to in a moment.

Keep in mind one thing: *`memcpy()` is evil!* It's a C construct you should never use in C++. `memcpy()` operates on bits and bytes, not on objects. At best, it just looks ugly and forces you to be concerned with things you shouldn't need to worry about, like the size of `TBar`. At worst, it fails to take into account what's actually being stored in the objects you're copying, leading to erroneous results, or even uglier code that takes the special cases into account. Never use `memcpy()` in C++ code. There are always better, more object-oriented ways to do the same thing.

So what's the right answer? Since the `TFoo` expects to delete the `TBars` it points to, we have to create new ones for it to point to. And we can create duplicates of the other `TFoo`'s `TBars` by using `TBar`'s copy constructor (remember from our introduction that every object has a copy constructor), so the correct solution (so far) would look like this:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    fBar1 = new TBar(*that.fBar1);
    fBar2 = new TBar(*that.fBar2);

    return *this;
}
```

Plugging the leaks

Of course, there's still a glaring error here: Remember that `fBar1` and `fBar2` are owning pointers. This means that `TFoo` is responsible for deleting them. Here, we've copied right over the top of these pointers without taking into account their former values. This'd be okay if we were writing a copy constructor, where we're guaranteed that `fBar1` and `fBar2` contain garbage, but it's not okay for an assignment operator. In the assignment operator, `fBar1` and `fBar2` are both valid pointers to `TBar` objects. If you just write over them, you now have two `TBar` objects in memory that *nobody* points to anymore (or at least, no one who points to them expects to have to delete them). This is a memory leak. Memory leaks won't cause your program to crash or produce wrong results, at least not right away. Instead, depending on how numerous and bad they are, they'll slowly degrade your program's performance. And if you run the program long enough, you'll run out of memory and it *will* crash.

So we have to delete the objects that we currently own before we can create new ones:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
```

```
    delete fBar1;
    fBar1 = new TBar(*that.fBar1);
    delete fBar2;
    fBar2 = new TBar(*that.fBar2);

    return *this;
}
```

Assigning yourself

Now we're beginning to get to something reasonable. But we're not quite there yet. Consider the following expression:

```
foo1 = foo1;
```

This might seem like kind of a silly example, but it does happen. Consider a setter function on some object where the variable being set is a `TFoo` and the other value is passed in. A caller knows he wants to set that variable to "x." The caller shouldn't have to check to see whether the value of that variable is already "x." If the value already *is* "x" and the caller doesn't check for this, look at what happens in our code: `this` and `that` refer to the same object, so by the time we get down to "`fBar1 = new TBar(*that.fBar1)`", `that.fBar1` is gone. "`delete fBar1`" also deleted `that.fBar1`. The call to `TBar`'s copy constructor will either crash because it's trying to access a deleted object, or it'll get away with that, create a brand-new `TBar`, and initialize it with the potentially random contents of raw memory. Worse, most of the time the data that had been in those two objects won't have been overwritten yet, so it'll probably work right 90% of the time and randomly fail the other 10%. This kind of bug is notoriously hard to track down.

There are many ways of coding around this, but the obvious answer is the best: just check at the top of the function to see whether you're assigning to yourself, and drop out if you are.

Of course, you can't just write

```
if (this == that)
```

because `this` and `that` are different types (`this` is a pointer and `that` is a reference). There are two ways of rectifying this: you can turn `this` into a reference, or turn `that` into a pointer. In other words, you could write

```
if (*this == that)
```

or you could write

```
if (this == &that)
```

Keep in mind that these two expressions don't do the same thing. The first example tests semantic equality, and the second example tests identity. In other words, "`if (*this == that)`" calls `TFoo`'s equality operator. If that object doesn't have one (it's *not* required or created for you), you're automatically hosed. If it does, it'll go through all the externally-visible data members of the two objects one by one looking for a difference, which is bound to be much slower than "`if (this != &that)`", which is a simple pointer comparison. "`if (this == &that)`" will return true only if `this` and `that` refer to the *same object*, not merely to two objects that "look the same." You'll occasionally do some unnecessary work this way, but it saves a lot of time and it protects you from crashing, which is really the point.

So (switching to "`!=`" for simplicity) our operator now looks like

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        delete fBar1;
        fBar1 = new TBar(*that.fBar1);
        delete fBar2;
        fBar2 = new TBar(*that.fBar2);
    }
    return *this;
```

```
}
```

Honoring your ancestry

We've also forgotten one other important detail. Remember the first line of our sample class definition:

```
class TFoo : public TSuperFoo {
```

`TFoo` is not a root class; it has a base class. This means we have to copy over the base class's data members too. If we were writing a copy constructor, we wouldn't generally have to worry about this, because the compiler will make sure our base class members are initialized before our constructor is called. But the compiler doesn't do anything like this for us with assignment operators; we have to do it ourselves.

The easiest way to do this is to call our superclass's assignment operator ourselves. You could do this by casting yourself to your base class:

```
*((TSuperFoo*)this) = that;
```

but it's much more readable to just call the inherited function by name:

```
TSuperFoo::operator=(that);
```

By the way, I've had a couple of people try

```
inherited::operator=(that);
```

Some of the older compilers on the Mac provided the `inherited` keyword, which I always liked. Unfortunately, it's not included in the C++ standard because it doesn't work well with multiple inheritance: in portable C++, you actually have to refer to your immediate superclass by name. So now we have

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        TSuperFoo::operator=(that);

        delete fBar1;
        fBar1 = new TBar(*that.fBar1);
        delete fBar2;
        fBar2 = new TBar(*that.fBar2);
    }
    return *this;
}
```

Cleaning up after yourself

We're still not really out of the woods here. The code above will work great... unless we encounter an error while trying to create one of our `TBar` objects. If we get an exception while creating a `TBar` object, the data member we're setting retains its old value, which now points to a deleted object. If we continue to use this `TFoo`, we'll probably eventually crash because of the dangling pointer. If we delete the `TFoo` in response to the exception, we'll probably blow sky high trying to double-delete the `TBar`.

If we know that the calling function will delete the object if assigning to it fails, we can just zero out the pointers after deleting the objects they refer to:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        TSuperFoo::operator=(that);

        delete fBar1;
        fBar1 = 0;
        fBar1 = new TBar(*that.fBar1);

        delete fBar2;
        fBar2 = 0;
```

```
        fBar2 = new TBar(*that.fBar2);
    }
    return *this;
}
```

Unfortunately, this only works if you're certain the calling function will catch the exception and delete the `TFoo`, or if `NULL` is a valid value for the `TBar` pointers. (Actually, if `NULL` is a valid value, you'll also have to check the incoming object's `fBar1` and `fBar2` for `NULL` before trying to new up new `TBars` from them.)

A better way to handle the problem is to make sure that the creation of a new object succeeds before you do anything to the variable you're assigning to. This way, if creation fails, you're still pointing to a perfectly good `TBar`— the assignment operation simply didn't have an effect. In fact, since we have two `TBars`, we should new up *both* of them before carrying out the assignment. This will ensure that the `TFoo` is always in an internally consistent state; either the whole assignment happened or none of it did:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        TSuperFoo::operator=(that);

        TBar* bar1;
        TBar* bar2;

        bar1 = new TBar(*that.fBar1);
        bar2 = new TBar(*that.fBar2);

        delete fBar1;
        fBar1 = bar1;
        delete fBar2;
        fBar2 = bar2;
    }
    return *this;
}
```

But there's a problem with this solution: Consider what happens if creation of `bar1` succeeds and creation of `bar2` fails. You'll exit with the actual object untouched, but what happens to the `TBar` pointed to by `bar1`? That's right; it leaks. In order to avoid this, you actually have to catch and re-throw the exception and delete `bar1` if it's been created. In order to tell if you're created `bar1`, you need to set it to `NULL` first, too. *And*, so we're really sure the assignment doesn't do anything until we know we've been able to create both of the new `TBars`, we can't call our inherited `operator=` function until after the try/catch block. So this all gets rather complicated:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        TBar* bar1 = 0;
        TBar* bar2 = 0;

        try {
            bar1 = new TBar(*that.fBar1);
            bar2 = new TBar(*that.fBar2);
        }
        catch (...) {
            delete bar1;
            delete bar2;
            throw;
        }

        TSuperFoo::operator=(that);
        delete fBar1;
        fBar1 = bar1;
        delete fBar2;
```

```
        fBar2 = bar2;
    }
    return *this;
}
```

Delegating our work

Of course, we're not really handling the exception here; we're just catching the exception to enable us to clean up properly. The try/catch block is a really ugly construct to have to use in this way. It'd be really nice if we could lose it.

One of the niceties of C++ exception handling is that destructors for any stack-based objects are guaranteed to be called even if the function that declares them exits prematurely with an exception or a return statement. We can take advantage of this behavior by having an object whose destructor deletes the object that would otherwise leak.

The new ANSI C++ standard provides us with just such an object: It's called `auto_ptr`. Using `auto_ptr`, we can write

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        auto_ptr<TBar> bar1 = new TBar(*that.fBar1);
        auto_ptr<TBar> bar2 = new TBar(*that.fBar2);

        TSuperFoo::operator=(that);
        delete fBar1;
        fBar1 = bar1.release();
        delete fBar2;
        fBar2 = bar2.release();
    }
    return *this;
}
```

The `release()` function gets rid of the `auto_ptr`'s reference to the object so that it won't delete the object in its destructor. So the `auto_ptr` will only delete the object it points to if we exit the function with an exception before getting to the `release()` call. Which is correct; the only things in this function which can fail are the constructor calls.

Taking full advantage

You've probably already guessed this part: We can actually utilize `auto_ptr` more fully than this. `auto_ptr` actually implements owning pointer semantics for us; if you assign to an `auto_ptr`, it deletes the object it points to before taking on the new pointer, and if an `auto_ptr` goes out of scope, it also automatically deletes the object it points to. So if we relaxed the rules of our exercise to allow us to redefine the class, we could redefine the class using `auto_ptr`:

```
class TFoo : public TSuperFoo {
    auto_ptr<TBar> fBar1;
    auto_ptr<TBar> fBar2;
    // method definitions...
};
```

Functions accessing the objects pointed to by `fBar1` and `fBar2` would look exactly the same as they did when `fBar1` and `fBar2` were regular pointers; `auto_ptr` defines its `*` and `->` operators to do the same thing as those for a regular pointer.

And now we can take full advantage of `auto_ptr` in our assignment operator:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        TSuperFoo::operator=(that);
```

```
        fBar1 = new TBar(*that.fBar1);
        fBar2 = new TBar(*that.fBar2);
    }
    return *this;
}
```

auto_ptr also takes care of deleting the objects previously pointed to by fBar1 and fBar2. One thing it doesn't automatically do for us, however, is create a new object to point to; we have to do that ourselves. If we just did

```
fBar1 = that.fBar1;
fBar2 = that.fBar2;
```

instead, we'd actually inadvertently affect that, when that's not what we want to do (in fact, by declaring that as const, we've *promised* not to affect it). This is because an auto_ptr's assignment operator passes ownership from one object to the other (this is the once exception to the rule about passing ownership that I mentioned earlier); in other words, it'd leave that.fBar1 and that.fBar2 with null pointers.

Actually, our auto_ptr solution doesn't do exactly the same thing as the previous example. If we want to make sure that all of the assignment happens or none of it, we still need the temporary variables. But this is a situation where the ownership-passing property of auto_ptr's assignment operator helps us:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        auto_ptr<TBar> bar1 = new TBar(*that.fBar1);
        auto_ptr<TBar> bar2 = new TBar(*that.fBar2);

        TSuperFoo::operator=(that);
        fBar1 = bar1;
        fBar2 = bar2;
    }
    return *this;
}
```

Here, if the second new operation fails, the first new TBar will be deleted by auto_ptr's destructor when we exit the function. But if both news succeed, the assignments will delete the objects fBar1 and fBar2 previously pointed to, and will also zero out bar1 and bar2 so that their destructors don't delete anything when we exit the function.

The other beauty of auto_ptr is its documentary value; if a pointer is declared as an auto_ptr, you know it's an owning pointer. If you consistently use auto_ptrs for all owning pointers and regular pointers only for aliasing pointers, the meanings of pointers are no longer ambiguous and you don't have to worry as much about naming conventions and documentation.

Of course, auto_ptr isn't available on all C++ compilers yet; if you're concerned about portability, don't use it. Do the assignment the first way I described, or make your own auto_ptr-like class. But if your compiler provides auto_ptr, and you're not worried about portability, it's definitely the way to go.

Miscellanea

I've gotten a couple questions in interviews that I probably should address briefly here. First, I had at least one applicant ask me why I could do things like

```
fBar1 = new TBar(*that.fBar1);
```

when fBar1 is a private member. The answer is that access control in C++ is done on an *class by class* basis, not on an instance-by-instance basis. Any instance of a class can access the private members of any other instance of the same class, so code like that shown in the example above is legal.

I also sometimes have people write code like

```
if (fBar1 != 0)
    delete fBar1;
```

This is actually unnecessary. The `delete` operator in C++ automatically performs this null check for you, so it's okay to delete a null pointer.

The *real* answer

Actually, I lied at the beginning of this article. There is an infinitely better solution to the problem as I stated it than the one we just worked our way to.

To see what I'm getting at, consider two points:

- If any other objects contain aliasing pointers to `TFoo`'s `TBar` objects, they will be invalid after you assign to the `TFoo`.
- Every object in a well-designed C++ system has a default constructor, a copy constructor, and an assignment operator.

That's right; you can perform the whole assignment just by calling *`TBar's`* assignment operator. That solution looks like this:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    TSuperFoo::operator=(that);
    *fBar1 = *(that.fBar1);
    *fBar2 = *(that.fBar2);

    return *this;
}
```

This is *so* much easier. So why didn't I start with this at the outset? There are several reasons:

- Going with the longer solution was a good way to explore many of the details of C++ memory management and syntax.
- The shorter solution won't work if `NULL` is a valid value for `fBar1` and `fBar2`. In this case, you'd have to use a hybrid solution: follow the longer example when you're copying an object into a null pointer, and the shorter example when you're copying an object into an object. (I'll leave this example as an exercise for the reader.)
- The shorter solution won't work if we want to transactionize the whole assignment. In other words, if we have a situation where *both* assignments have to complete successfully in order for our object to be in a consistent state, we can't use the simple solution because there's no way to roll back our changes to `fBar1` if assigning to `fBar2` throws an exception. You'd have to use the longer example. However, if the two data members are unrelated, or you know their assignment operators can't throw an exception, this solution is perfectly adequate.
- The shorter solution won't work if `TBar` is a polymorphic class. Let's take a closer look at this situation.

Handling polymorphism

I stated in the original problem that you could assume for the purposes of the exercise that `TBar` was a monomorphic class— that is, that `TBar` has no subclasses. If `TBar` is a polymorphic class—that is, if we know it has or can have subclasses—then the shorter solution won't work.

Let's pretend for a moment that `TBar` is an abstract class and that it has two concrete subclasses, `TDerivedBar1` and `TDerivedBar2` (yeah, I know I'm really creative with these names). All we have in our short example are

pointers to `TBar`. Each assignment above will call *`TBar's`* `operator=` function, *not* the `operator=` function of `TDerivedBar1` or `TDerivedBar2` (remember, `operator=()` isn't virtual). This means that any data members defined by the `TDerivedBar` classes won't be copied. This is called *slicing*, and it's something you have to watch out for in C++. You must always pay special attention to whether a class is polymorphic or monomorphic. Polymorphism imposes special restrictions on what you can do with your objects, and the compiler doesn't enforce these restrictions.

Of course, we could theoretically get around these problems by making `TBar`'s `operator=` function virtual. If `this->fBar1` is an instance of `TDerivedBar1`, you'll call `TDerivedBar1::operator=()` instead of `TBar::operator=()`. But if it's a `TDerivedBar2`, you're in trouble. `TDerivedBar1` isn't going to know what to do with `TDerivedBar2`'s members; it has nowhere to put them. You really want it to look like the object pointed to by `this->fBar1` has morphed from a `TDerivedBar2` to a `TDerivedBar1`. There's only one way to do this, and that's to delete the `TDerivedBar2` and new up a brand-new `TDerivedBar1`.

So our longer solution, where we delete the old objects and replace them with newly-created objects (either directly or with `auto_ptr`), is closer to the mark. But it won't work as written, either. Consider the line

```
fBar1 = new TBar(*that.fBar1);
```

If `TBar` is an abstract class, this will generate a compile-time error, because you're trying to create an instance of a class that can't be instantiated. If `TBar` is a concrete, but polymorphic, class, it'll compile, but you'll get slicing again: the `new` expression will only return an instance of `TBar`, even if the original object was a `TDerivedBar1`. At best this isn't a real copy and at worst it has incomplete and inconsistent state.

The C++ language doesn't provide a built-in way around this problem, unfortunately— if you need to copy an object polymorphically, you have to do it yourself. The typical way to do this is to define a virtual function called `clone()` and have every class that inherits `clone()` override it to call its own copy constructor.

```
class TBar {
    ...
    virtual TBar* clone() const = 0;
    ...
};

TDerivedBar1::clone() const
{
    return new TDerivedBar1(*this);
}

TDerivedBar2::clone() const
{
    return new TDerivedBar2(*this);
}
```

Once you've given all the classes in question a `clone()` method, you can go back and rewrite `TFoo`'s assignment operator properly:

```
TFoo&
TFoo::operator=(const TFoo& that)
{
    if (this != &that) {
        auto_ptr<TBar> bar1 = that.fBar1->clone();
        auto_ptr<TBar> bar2 = that.fBar2->clone();

        TSuperFoo::operator=(that);
        fBar1 = bar1;
        fBar2 = bar2;
    }
    return *this;
}
```

Counted pointers and other esoterica

There are a few other points about owning and aliasing pointers that I wanted to make.

The first is that aliasing pointers can be fragile. If you have other objects that have aliasing pointers to a `TFoo`'s `fBar1` object, they'll be bad after a call to `TFoo`'s assignment operator. This means you have to be careful that any aliases have a shorter lifetime than the object's owning pointer. Typically, aliases are short-lifetime objects: You obtain a aliasing pointer to a `TFoo`'s `fBar1` at the top of a function, use it to perform some operations on the `TBar`, and then let it go out of scope at the end of the function. If you were instead trying to keep the alias around for an indefinite period of time by making it a data member of some other object (you'd probably do this for performance or convenience), you're asking for trouble. The solution to this problem is almost always to make the persistent pointer point to something with a longer and more well-defined lifespan (for example, the `TFoo` that owns the `TBar` you're interested in).

Of course, there are situations where the relative lifetimes of a network of objects aren't clear, and therefore it isn't clear who should own whom, or which pointers can be kept around persistently. The solution to this problem is the same as the solution to the problem of spending too much time copying objects to maintain clear ownership semantics: reference counting.

In reference counting, each object keeps track of how many other objects point to it. These objects refer to the counted object through a *counted pointer,* which functions more or less like an aliasing pointer except that it notifies the object when it no longer refers to it (either because it now points to something else, or because it's going away). *Nobody* deletes the counted object; the counted object deletes *itself* when no one is left pointing to it.

The C++ standard doesn't provide canned `auto_ptr`-style objects that do reference counting, but most frameworks do. It can be expensive, especially in virtual-memory environments, though, so think twice before resorting to this solution.

An additional caveat is that if you use reference counting to improve performance in a situation where you'd otherwise be copying objects at the drop of a hat, you don't want to change the semantics of what you're doing. When you're sharing to avoid unnecessary copying, every object still expects to behave like he has his own copy of the shared object. That means the object can only be shared until someone tries to *change* it. At that point, the object trying to change it must create a copy of his own. This behavior is called *copy-on-write semantics.* Often, it's convenient to create a special counted-pointer class that enforces copy-on-write semantics transparently. This is almost never provided by a framework, however; you'd have to code it yourself.

Note, by the way, that in a garbage-collected environment like Java, much of this material is still relevant. It's true that the garbage collector frees you from having to worry about deleting things (much as reference counting does, but usually with less overhead), but that's it. It doesn't free you from having to make sure your object remains in a consistent state if the assignment fails halfway through. It doesn't free you from worrying about ownership semantics with resources other than memory blocks (such as disk files, window-system elements, or locks). And it doesn't free you from having to worry about when you need to duplicate an object and when an object can be shared. In fact, it exacerbates the last two problems. In C++, for example, an object can release its system resources in its destructor, so that all you have to worry about is deleting the object itself (and not even that if it's in a local variable), whereas you have to explicitly release them in Java. In C++, you can usually use "const" in interfaces to tell whether an object can change another object's state or not, where Java lacks "const," forcing the use of ad-hoc conventions similar to the ones we use in C++ for ownership semantics (or forcing a lot of extra copying). And in both languages, you have to worry about which objects referring to a shared object should see the change if the shared object changes (i.e., when do you give an object its own copy of another object, and when can you legitimately share?)

Garbage collection is definitely not a panacea.

Conclusion

Well, there you have it: a whirlwind tour of some of the finer points of C++ memory management, as seen through the lens of a simple assignment operator. C++ is a monster of a language, and writing good code in it takes some practice and some concentration. I hope these pointers help illuminate some of the more esoteric but important areas of C++ programming for you and help you to write better code.

And who knows? My team still has an opening.

Bibliography

Goldsmith, David, *Taligent's Guide to Designing Programs.* Reading, MA: Addison-Wesley, 1994. Affectionately known as "Miss Manners" around Taligent, this book is the definitive reference for low-level C++ details like the ones we've been discussing. Most of this material comes straight from this book. Every C++ programmer should read it.

Meyers, Scott, *Effective C++*. Reading, MA: Addison-Wesley, 1992. Similar to "Miss Manners," but a little more basic.

Coplien, James O., *Advanced C++ Programming Styles and Idioms.* Reading, MA: Addison-Wesley, 1992.

Alger, Jeff, *Secrets of the C++ Masters.* Cambridge, MA: Ap Professional, 1995. Excellent coverage of smart pointers.

*The more advanced programmers reading this will probably be able to come up with one or two better ways of doing this than this particular example. Stick with me; I'll get to them. I present this example as a good general solution to the problem that works with almost all C++ compilers.