Copy Constructors and Assignment Operators

Introduction

Unlike other object-oriented languages like Java, C++ has robust support for object deep-copying and assignment. You can choose whether to pass objects to functions by reference or by value, and can assign objects to one another as though they were primitive data types.

C++ handles object copying and assignment through two functions called *copy constructors* and *assignment operators*. While C++ will automatically provide these functions if you don't explicitly define them, in many cases you'll need to manually control how your objects are duplicated. This handout discusses copy constructors and assignment operators, including both high-level concepts and practical implementation techniques.

Assignment versus Initialization

Before discussing copy constructors and assignment operators, let's first discuss the subtle yet crucial difference between *assignment* and *initialization*.

Consider the following code snippet:

```
MyClass one;
MyClass two = one;
```

Here, the variable `two` is *initialized* to `one` because it is created as a copy of another variable. When `two` is created, it will go from containing garbage data directly to holding a copy of the value of `one` with no intermediate step.

However, if we rewrite the code as

```
MyClass one, two;
two = one;
```

Then `two` is *assigned* the value of `one`. Note that before we reach the line `two = one`, `two` already contains a value. This is the difference between assignment and initialization – when a variable is *created* to hold a specified value, it is being initialized, whereas when an *existing* variable is set to hold a new value, it is being assigned.

If you're ever confused about when variables are assigned and when they're initialized, you can check by sticking a `const` declaration in front of the variable. If the code still compiles, the object is being initialized. Otherwise, it's being assigned a new value.

Copy Constructors

Whenever you initialize an object, C++ will make the copy by invoking the class's *copy constructor*, a constructor that accepts another object of the same type as a parameter.

Copy constructors are invoked whenever:

1. A newly-created object is initialized to the value of an existing object.
2. An object is passed to a function as a non-reference parameter.
3. An object is returned from a function.

Let's consider each of these cases individually. The first case, when new objects are initialized to the value of existing objects, is easiest to see. For example, suppose you write the following code:

```
MyClass one;
MyClass two = one;
MyClass three = two;
```

Here, since `two` and `three` are being initialized to the values of `one` and `two`, respectively, C++ will invoke the copy constructors to initialize their values. Although it looks like you're assigning values to `two` and `three` using the = operator, since they're newly-created objects, the = indicates initialization, not assignment. In fact, the above code is equivalent to the more explicit initialization code below:

```
MyClass one;
MyClass two(one);   // Equivalent to the above code.
MyClass three(two); // Same here
```

For the second case, consider the following function:

```
void myFunction(MyClass parameter)
{
    // ... Work with parameter ...
}
```

If we write

```
MyClass mc;
myFunction(mc);
```

Then the variable `parameter` inside of `myFunction` will be initialized to a copy of `mc` using the copy constructor.

In the final case, suppose we have the following function:

```
MyClass myFunction()
{
    MyClass mc;
    return mc;
}
```

If we call `myFunction`, then C++ will create a new `myClass` object that's initialized to `mc` when `myFunction` returns. Thus while your code might act like it's transparently moving the object from inside of `myFunction` to the rest of your code, it's actually making a temporary copy.

Assignment Operators

While the copy constructor is used to set up a new version of an object that's a duplicate of another object, the *assignment operator* is used to overwrite the value of an already-created object with the contents of another class instance.

For example, the following code will invoke the assignment operator, not the copy constructor:

```
MyClass one, two;
two = one;
```

In this case, since `two` has already been initialized in the previous line, C++ will use the `MyClass` assignment operator to assign `two` the value of the variable `one`.

It can be tricky to differentiate between code using the assignment operator and code using the copy constructor. For example, if we rewrite the above code as

```
MyClass one;
MyClass two = one;
```

We are now invoking the copy constructor rather than the assignment operator. Always remember that the assignment operator is called only when assigning an existing object a new value. Otherwise, you're using the copy constructor.

What C++ Does For You

Unless you specify otherwise, C++ will automatically provide objects a basic copy constructor and assignment operator that simply invoke the copy constructors and assignment operators of all the class's data members. In many cases, this is exactly what you want. For example, consider the following class:

```
class MyClass
{
    public:
        /* Omitted. */
    private:
        int myInt;
        string myString;
};
```

Suppose you have the following code:

```
MyClass one;
MyClass two = one;
```

The line `MyClass two = one` will invoke the copy constructor for `MyClass`. Since we haven't explicitly provided our own copy constructor, C++ will simply initialize `two.myInt` to the value of `one.myInt` and `two.myString` to `one.myString`. Since `int` is a primitive and `string` has a well-defined copy constructor, this code is totally fine.

However, in many cases this is not the behavior you want. Let's consider the example of a class `CString` that acts as a wrapper for a C string. Suppose we define `CString` as shown here:

```
class CString
{
    public:
        CString();
        ~CString(); // Deallocates the stored string.
        // Note: No copy constructor or assignment operator
    private:
        char *theString;
};
```

Here, if we rely on C++'s default copy constructor or assignment operator, we'll run into trouble. For example, consider the following code:

```
CString one;
CString two = one;
```

Because we haven't provided a copy constructor, C++ will initialize `two.theString` to `one.theString`. Since `theString` is a `char *`, instead of getting a deep copy of the string, we'll end up with two pointers to the same C string. Thus changes to `one` will show up in `two` and vice-versa. This is dangerous, especially when the destructors for both `one` and `two` try to deallocate the memory for `theString`. In situations like these, you'll need to override C++'s default behavior by providing your own copy constructors and assignment operators.

The Rule of Three

There's a well-established C++ principle called the "rule of three" that almost always identifies the spots where you'll need to write your own copy constructor and assignment operator. If this were a math textbook, you'd probably see the rule of three written out like this:

Theorem: If a class has any of the following three member functions:

- Destructor
- Copy Constructor
- Assignment Operator

Then that class should have all three of those functions.

Corollary: If a class has a destructor, it should also have a copy constructor and assignment operator.

The rule of three holds because in almost all situations where you have any of the above functions, C++'s default behavior won't correctly manage your objects. In the above example with `CString`, this was the case because copying the `char *` pointer didn't actually duplicate the C string. Similarly, if you have a class holding an open file handle, making a shallow copy of the object might cause crashes further down the line as the destructor of one class closed the file handle, corrupting the internal state of all "copies" of that object.

Both C++ libraries and fellow C++ coders will expect that, barring special circumstances, all objects will correctly implement the three above functions, either by falling back on C++'s default versions or by explicitly providing correct implementations. Consequently, you *must* keep the rule of three in mind when designing classes or you will end up with insidious or seemingly untraceable bugs as your classes start to destructively interfere with each other.

Writing Copy Constructors

For the rest of this handout, we'll discuss copy constructors and assignment operators through a case study of the `DebugVector` class. `DebugVector` is a modified version of the CS106 `Vector` whose constructor and destructor write creation and destruction information to `cout`. That way, if you're writing a program that you think might be leaking memory, you can check the program output to make sure that all your `DebugVector`s are begin cleaned up properly.

The class definition for `DebugVector` looks like this:

```
template<typename T>
class Vector
{
    public:
        DebugVector();
        DebugVector(const DebugVector &other); // Copy constructor
        DebugVector &operator =(const DebugVector &other); // Assn. operator
        ~DebugVector();

        /* Other member functions. */
    private:
        T *array;
        int allocatedLength;
        int logicalLength;
        static const int BASE_SIZE = 16;
};
```

The syntax for the copy constructor and assignment operator might look a bit foreign. The copy constructor is simply a class constructor that accepts as a parameter a `const` reference to another instance of the class. The assignment operator, however, is a special type of function called an *overloaded operator*, a topic we'll cover in more detail next week. For now, you should just take on faith that the function named `operator =` will redefine what it means to use the = operator in code.

The `DebugVector` is internally implemented as a dynamically-allocated array of elements. We have two data members – `allocatedLength` and `logicalLength` – that track the allocated size of the array and the number of elements stored in it, respectively. It also has a class constant `BASE_SIZE` that represents the base size of the allocated array.

The `DebugVector` constructor is defined as

```
template<typename T>
DebugVector<T>::DebugVector() : array(new T[BASE_SIZE]),
    allocatedLength(BASE_SIZE), logicalLength(0)
{
    // Log the creation using some functions from <ctime>
    time_t currentTime;
    time(&currentTime); // Fill in the current time.
    cout << "DebugVector created: " << ctime(&currentTime) << endl;
}
```

Note that we're initializing `array` to point to a dynamically-allocated array of `BASE_SIZE` elements in the initializer list.

Similarly, the `DebugVector` destructor is

```
template<typename T>
DebugVector<T>::~DebugVector()
{
    delete [] array;
    array = NULL; // Just to be safe
    logicalLength = allocatedLength = 0;

    time_t currentTime;
    time(&currentTime);
    cout << "Destructor invoked: " << ctime(&currentTime) << endl;
}
```

Now, let's write the copy constructor. We know that we need to match the prototype given in the class definition, so we'll write that part first:

```
template<typename T>
DebugVector<T>::DebugVector(const DebugVector &other)
{
    // Our code goes here.
}
```

Inside the copy constructor, we'll need to do two things. First, we'll need to initialize the object so that we're holding a deep copy of the other `DebugVector`. Second, we'll need to log the creation information since we're instantiating a new `DebugVector`. Let's first initialize the object, then handle the logging later.

As with a regular constructor, with a copy constructor we should try to initialize as many instance variables as possible in the initializer list, both for readability and speed. In the case of `DebugVector`, while we can't completely set up the data members in the initializer list, we can still copy over the value of `logicalLength` and `allocatedLength`. For this handout, however, we'll initialize these variables inside the body of the constructor, since `logicalLength` and `allocatedLength` are both primitives and thus don't get a performance boost from the initializer list. Thus our copy constructor will look something like this:

```
template<typename T>
DebugVector<T>::DebugVector(const DebugVector &other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;
    // Incomplete, but on the right track
}
```

Note that although `other`'s `logicalLength` and `allocatedLength` are private variables, we're allowed to access them freely. This is sometimes known as “sibling access” and is true of any member function, not just the copy constructor.

Now, we'll make a deep copy of the other `DebugVector`'s elements by allocating a new array that's the same size as `other`'s and then copying the elements over. The code looks something like this:

```
template<typename T>
DebugVector<T>::DebugVector(const DebugVector &other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;
    array = new T[allocatedLength];
    for(int i = 0; i < logicalLength; i++)
        array[i] = other.array[i];
    // Still need to log everything.
}
```

Interestingly, since `DebugVector` is a template, it's unclear what the line `array[i] = other.array[i]` will actually do. If we're storing primitive types, then the line will simply copy the values over, but if we're storing objects, the line calls the class's assignment operator. Notice that in both cases the object will be correctly copied over. This is one of driving forces behind defining copy constructors and assignment operators, since template code can assume that expressions like `object1 = object2` will be meaningful.

Finally, we need to log the creation of the new `DebugVector`. If you'll notice, the code for logging the `DebugVector`'s creation is already written for us in the constructor. Unfortunately, unlike other object-oriented languages, in C++, a class with multiple constructors can't invoke one constructor from the body of another. To avoid code duplication, we'll therefore move the logging code into a separate private member function called `logCreation` that looks like this:

```
template<typename T>
void DebugVector<T>::logCreation()
{
    time_t currentTime;
    time(&currentTime); // Fill in the current time.
    cout << "DebugVector created: " << ctime(&currentTime) << endl;
}
```

We then rewrite the default constructor for `DebugVector` to call `logCreation`, as shown below:

```
template<typename T>
DebugVector<T>::DebugVector() : array(new T[BASE_SIZE]),
    allocatedLength(BASE_SIZE), logicalLength(0)
{
    logCreation();
}
```

And finally, we insert a call to `logCreation` into the copy constructor, yielding the final version, which looks like this:

```
template<typename T>
DebugVector<T>::DebugVector(const DebugVector &other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;
    array = new T[allocatedLength];
    for(int i = 0; i < logicalLength; i++)
        array[i] = other.array[i];
    logCreation();
}
```

Writing Assignment Operators

We've now successfully written a deep-copying copy constructor for our `DebugVector` class. Unfortunately, writing an assignment operator is significantly more involved than writing a copy constructor. C++ is designed to give you maximum flexibility when designing an assignment operator, and thus won't alert you if you've written a syntactically legal assignment operator that is completely incorrect. For example, consider this legal but incorrect assignment operator for an object of type `MyClass`:

```
void MyClass::operator =(const MyClass &other)
{
    cout << "Sorry, Dave, I can't copy that object." << endl;
}
```

Here, if we write code like this:

```
MyClass one, two;
two = one;
```

Instead of making `two` a deep copy of `one`, instead we'll get a message printed to the screen and `two` will remain unchanged. This is one of the dangers of operator overloading – code that looks like it does one thing can instead do something totally different. This next section will discuss how to correctly implement an assignment operator by starting with invalid code and progressing towards a correct, final version.

Let's start off with a simple but incorrect version of the assignment operator for `DebugVector`:

```
/* Many major mistakes here. */
template<typename T>
void DebugVector<T>::operator =(const DebugVector &other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;
    array = new T[allocatedLength];
    for(int i = 0; i < logicalLength; i++)
        array[i] = other.array[i];
}
```

The most serious error here is the line `array = new T[allocatedLength]`. When the assignment operator is invoked, this `DebugVector` is holding on to its own array of elements. However, with this line, we're orphaning our old array and leaking a good deal of memory. To fix this, before we make this object a copy of the one specified by the parameter, we'll take care of the necessary deallocations.

If you'll notice, we've already written some of the necessary cleanup code in the destructor. Rather than rewriting this code, we'll decompose out the generic cleanup code into a `clear` function, as shown here:

```
template<typename T>
void DebugVector<T>::clear()
{
    delete [] array;
    array = NULL;
    logicalLength = allocatedLength = 0;
}
```

We can then rewrite the destructor as

```
template<typename T>
DebugVector<T>::~DebugVector()
{
    clear();
    time_t currentTime;
    time(&currentTime);
    cout << "Destructor invoked: " << ctime(&currentTime) << endl;
}
```

And we can insert this call to `clear` into our assignment operator as follows:

```
/* Still are some errors here. */
template<typename T>
void DebugVector<T>::operator =(const DebugVector &other)
{
    clear();
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;
    array = new T[allocatedLength];
    for(int i = 0; i < logicalLength; i++)
        array[i] = other.array[i];
}
```

Along the same lines, you might have noticed that all of the code after the call to `clear` is exactly the same code we wrote inside the body of the copy constructor. This isn't a coincidence – in fact, in most cases you'll have a good deal of overlap between the assignment operator and copy constructor. Since we can't invoke our own copy constructor, instead we'll decompose the copying code into a member function called `copyOther` as follows:

```
template<typename T>
void DebugVector<T>::copyOther(const DebugVector &other)
{
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;
    array = new T[allocatedLength];
    for(int i = 0; i < logicalLength; i++)
        array[i] = other.array[i];
}
```

Now we can rewrite the copy constructor as

```
template<typename T>
DebugVector<T>::DebugVector(const DebugVector &other)
{
    copyOther(other);
    logCreation();
}
```

And the assignment operator as

```
/* Not quite perfect yet. */
template<typename T>
void DebugVector<T>::operator =(const DebugVector &other)
{
    clear();
    copyOther(other);
}
```

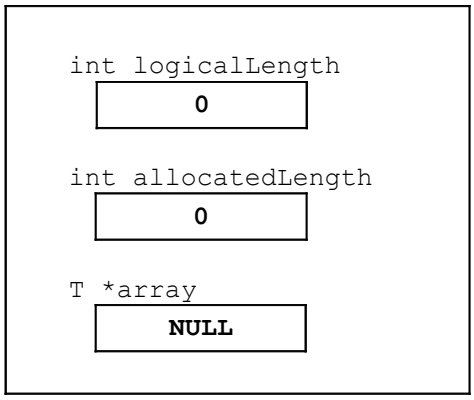
This simplifies the copy constructor and assignment operator and highlights the general pattern of what the two functions should do. With a copy constructor, you'll copy the contents of the other object, then perform any remaining initialization. With an assignment operator, you'll clear out the current object, then copy over the data from another object.

However, we're still not done yet. There are two more issues we need to fix with our current implementation of the assignment operator. The first one has to do with *self-assignment*. Consider, for example, the following code:

```
MyClass one;
one = one;
```

While this code might seem a bit silly, cases like this come up frequently when accessing elements indirectly through pointers or references. Unfortunately, with our current `DebugVector` assignment operator, this code will lead to unusual runtime behavior. To see why, let's trace out the state of our object when its assignment operator is invoked on itself.

At the start of the assignment operator, we call `clear` to clean out the object for the copy. During this call to `clear`, we deallocate the memory associated with the object and set both `logicalLength` and `allocatedLength` to zero. Thus our current object looks something like this:



Normally this isn't a problem, but remember that we're self-assigning, which means that the object referenced by `other` is the same object referenced by the `this` pointer. Consequently, since we erased all the contents of the current object, we also erased the contents of the `other` object. When we get to the code in `copyOther`, we'll end up duplicating an empty object. The result is that our self-assignment is effectively a glorified call to `clear`.

When writing assignment operators, you absolutely must ensure that your code correctly handles self-assignment. While there are many ways we can do this, perhaps the simplest is to simply check to make sure that the object to copy isn't the same object pointed at by the `this` pointer. The code for this logic looks like this:

```
/* Not quite perfect yet - there's one more error */
template<typename T>
void DebugVector<T>::operator =(const DebugVector &other)
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
}
```

Note that we check `if(this != &other)`. That is, we are comparing *the addresses* of the current object and the parameter. This will determine whether or not the object we're copying is exactly the same object as the one we're working with. Note that we did *not* write `if(*this != other)`, since this would do a semantic comparison between the objects, something we'll cover next week when discussing operator overloading. One last note is that code like this isn't required in the copy constructor, since we can't pass an object as a parameter to its own copy constructor.

There's one final bug we need to sort out, and it has to do with how we're legally allowed to use the = operator. Consider, for example, the following code:

```
MyClass one, two, three;
three = two = one;
```

Since our current assignment operator does not return a value, the above code will not compile. Thus we need to change to our assignment operator so that it returns a value that can be used in future assignments.

The final version of our assignment operator is thus

```
/* The CORRECT version. */
template<typename T>
DebugVector<T> & DebugVector<T>::operator =(const DebugVector &other)
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}
```

Note that we're returning `*this`, the value of the current object. Since the return type is a `DebugVector<T> &`, we're actually returning a reference to the current object, rather than a deep copy. Since the expression `one = two = three` is equivalent to `one = (two = three)`, this will work out correctly.

One General Pattern

Although each class is different, in many cases the default constructor, copy constructor, assignment operator, and destructor will share a general pattern. Here is one possible skeleton you can fill in to get your copy constructor and assignment operator working.

```
MyClass::MyClass() : /* Fill in initializer list. */
{
    additionalInit();
}

MyClass::MyClass(const MyClass &other)
{
    copyOther(other);
    additionalInit(); // Don't set any data members here if possible.
}
MyClass &MyClass::operator =(const MyClass &other)
{
    if(this != &other)
    {
        clear();
        // Note: When we cover inheritance, there's one more step here.
        copyOther(other);
    }
    return *this;
}

MyClass::~MyClass()
{
    clear();
    additionalClosedown();
}
```

Semantic Equivalence and `copyOther` Strategies

Consider the following code snippet:

```
DebugVector<int> one;
DebugVector<int> two = one;
```

Here, we know that `two` is a copy of `one`, so the two objects should behave identically to one another. For example, if we access an element of `one`, we should get the same value as if we had accessed an element of `two` and vice-versa. However, while `one` and `two` are indistinguishable from each other in terms of functionality, their memory representations are not identical because `one` and `two` point to two different dynamically-allocated arrays. This raises the distinction between *semantic equivalence* and *bitwise equivalence*. Two objects are said to be bitwise equivalent if they have identical binary representations in memory. For example, any two `int`s with the value 137 are bitwise equivalent, and if we define a `pointT` struct as a pair of `int`s, any two `pointT`s holding the same location will be bitwise equivalent. Two objects are semantically equivalent if, like `one` and `two`, their behaviors are identical.

When writing a copy constructor and assignment operator, you attempt to convert an object into a semantically equivalent copy of another object. Consequently, when duplicating certain objects, you are free to pick any copying strategy that creates a semantically-equivalent copy of the source object.

In the preceding section, we outlined one possible implementation strategy for a copy constructor and assignment operator that references a function called `copyOther`. While in the case of the `DebugVector` it was relatively easy to come up with a working `copyOther` implementation, when working with more complicated objects, it can be difficult to devise a working `copyOther`. For example, consider the following class, which encapsulates an unordered linked list:

```
template<typename T> class ListSet
{
public:
    ListSet();
    ListSet(const ListSet &other);
    ListSet &operator =(const ListSet &other);
    ~ListSet();
    void insert(const T &toAdd);
    bool contains(const T& toFind) const;
private:
    struct cellT {
        T data;
        cellT *next;
    };
    cellT *list
    void copyOther(const ListSet &other);
};
```

This simple `ListSet` class exports two simple functions, `insert` and `contains`, that insert an element into the list and determine whether the list contains an element. If you'll notice, the `add` function gives no guarantees about where the element will be inserted, and the `contains` function ignores ordering, so the lists `{0, 1, 2, 3, 4}` and the lists `{4, 3, 2, 1, 0}` are semantically equivalent. Consider two different implementations of `copyOther`:

```
/* Duplicate the list as it exists in the original ListSet. */
template<typename T>
void ListSet<T>::copyOther(const ListSet &other)
{
    /* Keep track of what the current linked list cell is. */
    cellT **current = &list;

    /* Iterate over the source list. */
    for(cellT *source = other.list; source != NULL; source = source->next)
    {
        /* Duplicate the cell. */
        *current = new cellT;
        (*current)->data = source->data;
        (*current)->next = NULL;

        /* Advance to next element. */
        current = &((*current)->next);
    }
}

/* Duplicate list in reverse order of original ListSet */
template<typename T>
void ListSet<T>::copyOther(const ListSet &other)
{
    for(cellT *source = other.list; source != NULL; source = source->next)
    {
        cellT *newNode = new cellT;
        newNode->data = source->data;
        newNode->next = list;
        list = newNode;
    }
}
```

As you can see, the second version of this function is much, *much* cleaner than the first. There are no address-of operators floating around, so everything is expressed in terms of simpler pointer operations. But while the second version is cleaner than the first, it duplicates the list in reverse order. Is this a problem? The answer is no. Recall that because we structured the class interface so that ordering doesn't matter, so long as the original object and the duplicate object contain the same elements in *some* order, they will be semantically equivalent, and from the class interface we would be unable to distinguish the original and source objects.

Consider this final version of the `copyOther` function for the `ListSet`, which duplicates the elements using the `ListSet` public interface:

```
/* Duplicate list using the add function */
template<typename T>
void ListSet<T>::copyOther(const ListSet &other)
{
    for(cellT *source = other.list; source != NULL; source = source->next)
        add(list->data);
}
```

This version of `copyOther` is unquestionably the cleanest. If you'll notice, it doesn't matter exactly how `add` inserts elements into the list (indeed, `add` could insert the elements at random positions), but we're guaranteed that at the end of the `copyOther` call, the receiving object will be semantically equivalent to the source object.

Disabling Copying

In CS106B/X we provide you the `DISALLOW_COPYING` macro, which causes compile-time errors if you try to assign or copy objects of the specified type. `DISALLOW_COPYING`, however, is not a standard C++ feature. Without using the CS106 library, how can we replicate the functionality?

We can't prevent object copying by simply not defining a copy constructor and assignment operator. All this will do is have C++ provide its own default version of these two functions, which is not at all what we want. To solve this problem, provide an assignment operator and copy constructor, but declare them private so that class clients can't access them. For example:

```
class CannotBeCopied
{
    public:
        CannotBeCopied();
        /* Other member functions. */
    private:
        CannotBeCopied(const CannotBeCopied &other);
        CannotBeCopied &operator = (const CannotBeCopied &other);
};
```

Now, if we write code like this:

```
CannotBeCopied one;
CannotBeCopied two = one;
```

We'll get a compile-time error on the second line because we're trying to invoke the copy constructor, which has been declared private. We'll get similar behavior when trying to use the assignment operator.

Note that when using this trick, you don't need to provide a body for either member function. Since class clients can't invoke the member functions, C++ won't need any code for them.

More to Explore

While the above method for writing an assignment operator is completely valid, there is another common way to write an assignment operator called "copy-and-swap." First, define a `swap` member function that exchanges the contents of two class instances by swapping all the instance variables (don't have `swap` make a deep copy, or you'll end up with unreasonably slow copying). Then, have the assignment operator construct a temporary copy of the object to duplicate and then `swap` the current object and the temporary copy. That way, the copy constructor handles the object duplication, and all the current object's data members will be cleaned up by temporary object's destructor.

Practice Problems

The only way to learn copy constructors and assignment operators is to play around with them to gain experience. Here are some practice problems and thought questions to get you started:

1. It is syntactically illegal to declare a copy constructor that accepts its parameter by value. For example, given a class `MyClass`, you cannot declare its copy constructor as `MyClass::MyClass(MyClass other)`. Why not?
2. It is, however, syntactically legal to declare an assignment operator that accepts its parameter by value. Why?
3. In the next handout, we'll see the class `CString` used as a case study for *conversion constructors*. `CString` is simply a class that wraps a C string, storing as data members the `char *` pointer to the C string and an `int` length field. Write a copy constructor and assignment operator for `CString`.
4. Suppose you're implementing the `Lexicon` class and, for efficiency reasons, you decide to store the words in a sorted `vector` of C strings (`vector<char *>`). Assume that the constructor has been provided to you and that it correctly initializes the `vector` by filling it with strings dynamically allocated from `new[]`. Write the copy constructor, assignment operator, and destructor for `Lexicon`. *(Hint: When the `vector` destructor invokes, it will **not** call `delete []` on all of the internally stored `char *`s, so you will have to do this yourself)*
5. If the above `Lexicon` used a `vector<string>` instead of a `vector<char *>`, would you need any of the functions mentioned in the rule of three?
6. What happens if you forget to copy a data member in a copy constructor or assignment operator?
7. Give an example of a class where bitwise equivalence and semantic equivalence are identical.
8. Give an example of a class where bitwise equivalence and semantic equivalence are *not* identical. *(Hint: where are the objects stored in memory?)*