Chapter 11: Resource Management

This chapter is about two things – putting away your toys when you're done with them, and bringing enough of your toys for everyone to share. These are lessons you (hopefully!) learned in kindergarten which happen to pop up just about everywhere in life. We're supposed to clean up our own messes so that they don't accumulate and start to interfere with others, and try to avoid hogging things so that others don't hurt us by trying to take those nice things away from us.

The focus of this chapter is how to play nice with others when it comes to *resources*. In particular, we will explore two of C++'s most misunderstood language features, the *copy constructor* and the *assignment operator*. By the time you're done reading this chapter, you should have a much better understanding of how to manage resources in ways that will keep your programs running and your fellow programmers happy. The material in this chapter is somewhat dense, but fear not! We'll make sure to go over all of the important points neatly and methodically.

Consider the STL `vector`. Internally, `vector` is backed by a dynamically-allocated array whose size grows when additional space is needed for more elements. For example, a ten-element `vector` might store those elements in an array of size sixteen, increasing the array size if we call `push_back` seven more times. Given this description, consider the following code:

```
vector<int> one(kNumInts);
for(size_t k = 0; k < one.size(); ++k)
    one.push_back(int(k));

vector<int> two = one;
```

In the first three lines, we fill `one` with the first `kNumInts` integers, and in the last line we create a new `vector` called `two` that's a copy of `one`. How does C++ know how to correctly copy the data from `one` into `two`? It can't simply copy the pointer to the dynamically-allocated array from `one` into `two`, since that would cause `one` and `two` to share the same data and changes to one `vector` would show up in the other. Somehow C++ is aware that to copy a `vector` it needs to dynamically allocate a new array of elements, then copy the elements from the source to the destination. This is not done by magic, but by two special functions called the *copy constructor* and the *assignment operator*, which control how to copy instances of a particular class.

Before discussing the particulars of the copy constructor and assignment operator, we first need to dissect exactly how an object can be copied. In order to copy an object, we first have to answer an important question – where do we put the copy? Do we store it in a new object, or do we reuse an existing object? These two options are fundamentally different from one another and C++ makes an explicit distinction between them. The first option – putting the copy into a new location – creates the copy by *initializing* the new object to the value of the object to copy. The second – storing the copy in an existing variable – creates the copy by *assigning* the existing object the value of the object to copy. What do these two copy mechanisms look like in C++? That is, when is an object initialized to a value, and when is it assigned a new value?

In C++, initialization can occur in three different places:

1. *A variable is created as a copy of an existing value.* For example, suppose we write the following code:

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

Here, since `two` is told to hold the value of `one`, C++ will *initialize* `two` as a copy of `one`. Although it looks like we're assigning a value to `two` using the = operator, since it is a newly-created object, the = indicates initialization, not assignment. In fact, the above code is equivalent to the more explicit initialization code below:

```
MyClass one;
MyClass two(one);    // Identical to above.
```

This syntax makes more clear that `two` is being created as a copy of `one`, indicating initialization rather than assignment.

2. *An object is passed by value to a function.* Consider the following function:

```
void MyFunction(MyClass arg) {
     /* ... */
}
```

If we write

```
MyClass mc;
MyFunction(mc);
```

Then the function `MyFunction` somehow has to set up the value of `arg` inside the function to have the same value as `mc` outside the function. Since `arg` is a new variable, C++ will *initialize* it as a copy of `mc`.

3. *An object is returned from a function by value.* Suppose we have the following function:

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

When the statement `return mc` executes, C++ needs to return the `mc` object from the function. However, `mc` is a local variable inside the `MyFunction` function, and to communicate its value to the `MyFunction` caller C++ needs to create a copy of `mc` before it is lost. This is done by creating a temporary `MyClass` object for the return value, then *initializing* it to be a copy of `mc`.

Notice that in all three cases, initialization took place because some new object was created as a copy of an existing object. In the first case this was a new local variable, in the second a function parameter, and in the third a temporary object.

Assignment in C++ is much simpler than initialization and only occurs if an existing object is explicitly assigned a new value. For example, the following code will *assign* `two` the value of `one`, rather than *initializing* `two` to `one`:

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

It can be tricky to differentiate between initialization and assignment because in some cases the syntax is almost identical. For example, if we rewrite the above code as

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

`two` is now initialized to `one` because it is declared as a new variable. Always remember that the assignment only occurs when giving an existing object a new value.

Why is it important to differentiate between assignment and initialization? After all, they're quite similar; in both cases we end up with a new copy of an existing object. However, assignment and initialization are fundamentally different operations. When *initializing* a new object as a copy of an existing object, we simply need to copy the existing object into the new object. When *assigning* an existing object a new value, the existing object's value ceases to be and we must make sure to clean up any resources the object may have allocated before setting it to the new value. In other words, initialization is a straight copy, while assignment is cleanup followed by a copy. This distinction will become manifest in the code we will write for the copy functions later in this chapter.

Copy Functions: Copy Constructors and Assignment Operators

Because initialization and assignment are separate tasks, C++ handles them through two different functions called the *copy constructor* and the *assignment operator*. The copy constructor is a special constructor responsible for initializing new class instances as copies of existing instances of the class. The assignment operator is a special function called an *overloaded operator* (see the chapter on operator overloading for more details) responsible for assigning the receiver object the value of some other instance of the object. Thus the code

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

will initialize `two` to `one` using the copy constructor, while the code

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

will assign `one` to `two` using the assignment operator.

Syntactically, the copy constructor is written as a one-argument constructor whose parameter is another instance of the class accepted by reference-to-`const`. For example, given the following class:

```
class MyClass {
public:
    MyClass();
    ~MyClass();

    /* ... */
};
```

The copy constructor would be declared as follows:

```
class MyClass {
public:
    MyClass();
    ~MyClass();

    MyClass(const MyClass& other); // Copy constructor

    /* ... */
};
```

The syntax for the assignment operator is substantially more complex than that of the copy constructor because it is an overloaded operator; in particular, `operator =`. For reasons that will become clearer later in the chapter, the assignment operator should accept as a parameter another instance of the class by reference-to-`const` and should return a non-`const` reference to an object of the class type. For a concrete example, here's the assignment operator for `MyClass`:

```
class MyClass {
public:
    MyClass();
    ~MyClass();

    MyClass(const MyClass& other); // Copy constructor
    MyClass& operator = (const MyClass& other); // Assignment operator
    /* ... */
};
```

We'll defer discussing exactly why this syntax is correct until later, so for now you should take it on faith.

What C++ Does For You

Unless you specify otherwise, C++ will automatically provide any class you write with a basic copy constructor and assignment operator that 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 DefaultClass {
public:
    /* ... */

private:
    int myInt;
    string myString;
};
```

Suppose you have the following code:

```
DefaultClass one;
DefaultClass two = one;
```

The line `DefaultClass two = one` will invoke the copy constructor for `DefaultClass`. Since we haven't explicitly provided our own copy constructor, C++ will 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 `Vector` that acts as a wrapper for a dynamic array. Suppose we define `Vector` as shown here:

```
class Vector {
public:
    Vector();
    ~Vector();
    /* Note: No copy constructor or assignment operator */

    /* ... */

private:
    int* elems;
    /* ... */
};
```

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

```
Vector one;
Vector two = one;
```

Because we haven't provided a copy constructor, C++ will initialize `two.elems` to `one.elems`. Since `elems` is an `int*`, instead of getting a deep copy of the elements, we'll end up with two pointers to the same array. 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 `elems`. In situations like these, you'll need to override C++'s default behavior by providing your own copy constructors and assignment operators.

There are a few circumstances where C++ does not automatically provide default copy constructors and assignment operators. If your class contains a reference or `const` variable as a data member, your class will not automatically get an assignment operator. Similarly, if your class has a data member that doesn't have a copy constructor or assignment operator (for example, an `ifstream`), your class won't be copyable. There is one other case involving inheritance where C++ won't automatically create the copy functions for you, and in the chapter on inheritance we'll see how to exploit this to disable copying.

The Rule of Three

There's a well-established C++ principle called the "rule of three" that identifies most 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 *(The Rule of Three)*: 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 `Vector`, this is the case because copying the `elems*` pointer doesn't actually duplicate the elements array. 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 chapter, we'll discuss copy constructors and assignment operators through a case study of a `Vector` class, a generalization of the above `Vector` which behaves similarly to the STL `vector`. The class definition for `Vector` looks like this:

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

    typedef T* iterator;
    typedef const T* const_iterator;

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    /* ... other member functions ... */
private:
    T* array;
    size_t allocatedLength;
    size_t logicalLength;
    static const size kStartSize = 16;
};
```

Internally, `Vector` is implemented as a dynamically-allocated array of elements. Two data members, `allocatedLength` and `logicalLength`, track the allocated size of the array and the number of elements stored in it, respectively. `Vector` also has a class constant `kStartSize` that represents the default size of the allocated array.

The `Vector` constructor is defined as

```
template <typename T> Vector<T>::Vector() {
    allocatedLength = kStartSize;
    logicalLength = 0;
    array = new T[allocatedLength];
}
```

Similarly, the `Vector` destructor is

```
template <typename T> Vector<T>::~Vector() {
    delete [] array;
}
```

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> Vector<T>::Vector(const Vector& other) {
    /* ... */
}
```

Inside the copy constructor, we need to initialize the object so that we're holding a deep copy of the other `Vector`. This necessitates making a full deep-copy of the other `Vector`'s array, as well as copying over information about the size and capacity of the other `Vector`. This second step is relatively straightforward, and can be done as follows:

```
template <typename T> Vector<T>::Vector(const DebugVector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    /* ... */
}
```

Note that this implementation of the copy constructor sets `logicalLength` to `other.logicalLength` and `allocatedLength` to `other.allocatedLength`, even though `other.logicalLength` and `other.allocatedLength` explicitly reference private data members of the `other` object. This is legal because `other` is an object of type `Vector` and the copy constructor is a member function of `Vector`. A class can access both its private fields and private fields of other objects of the same type. This is called *sibling access* and is true of any member function, not just the copy constructor. If the copy constructor were not a member of `Vector` or if `other` were not a `Vector`, this code would not be legal.

Now, we'll make a deep copy of the other `Vector`'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> Vector<T>::Vector(const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    for(size_t i = 0; i < logicalLength; ++i)
        array[i] = other.array[i];
}
```

Interestingly, since `Vector` 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 invokes 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.

An alternative means for copying data over from the other object uses the STL `copy` algorithm. Recall that `copy` takes three parameters – two delineating an input range of iterators and one denoting the beginning of an output range – then copies the specified iterator range to the destination. Although designed to work on iterators, it is possible to apply STL algorithms directly to ranges defined by raw C++ pointers. Thus we could rewrite the copy constructor as follows:

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

Here, the range spanned by `other.begin()` and `other.end()` is the entire contents of the other `Vector`, and `array` is the beginning of the newly-allocated data we've reserved for this `Vector`. I personally find this syntax preferable to the explicit `for` loop, since it increases readability.

At this point we have a complete and correct implementation of the copy constructor. The code for this constructor is not particularly dense, and it's remarkably straightforward. In some cases, however, it can be a bit trickier to write a copy constructor. We'll see some of these cases later in the chapter.

Writing Assignment Operators

We've now successfully written a copy constructor for our `Vector` 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 << "I'm sorry, Dave.  I'm afraid 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 a poorly-written assignment operator – code that looks like it does one thing can instead do something totally different. This section discusses 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 `Vector`. Intuitively, since both the copy constructor and the assignment operator make a copy of another object, we might consider implementing the assignment operator by naively copying the code from the copy constructor into the assignment operator. This results in the following (incorrect!) version of the assignment operator:

```
/* Many major mistakes here.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

This code is based off the copy constructor, which we used to initialize the object as a copy of an existing object. Unfortunately, this code contains a substantial number of mistakes that we'll need to correct be-

fore we end up with the final version of the function. Perhaps the most serious error here is the line `array = new T[allocatedLength]`. When the assignment operator is invoked, this `Vector` already holds its own array of elements. This line therefore orphans the old array and leaks 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. This is shown here:

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

```
/* Many major mistakes here.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    delete [] array;

    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

At this point, we can make a particularly useful observation. If you'll notice, the cleanup code to free the existing array is identical to the code for the destructor, which has the same task. This is no coincidence. In general, when writing an assignment operator, the assignment operator will need to free all of the resources acquired by the object, much in the same way that the destructor must. To avoid unnecessary code duplication, we can factor out the code to free the `Vector`'s resources into a helper function called `clear()`, which is shown here:

```
template <typename T> void Vector<T>::clear() {
    delete [] array;
}
```

We can then rewrite the destructor as

```
template <typename T> Vector<T>::~Vector() {
    clear();
}
```

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

```
/* This code still has errors.  Do not use it as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    clear();

    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

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 directly (or *any* constructor, for that matter), instead we'll decompose the copying code into a member function called `copyOther` as follows:

```
template <typename T> void Vector<T>::copyOther(const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

Now we can rewrite the copy constructor as

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

And the assignment operator as

```
/* Not quite perfect yet.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& 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 simply copy the contents of the other object. With an assignment operator, you'll clear out the receiver 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, possibly including a crash. 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. We then invoke the `copyOther` function to set the current object to be a copy of the receiver object. Unfortunately, things don't go quite as expected. Because we're assigning the object to itself, the parameter to the assignment operator is the receiver object itself. This means that when we called `clear` trying to clean up the resources associated with the receiver object, we also cleaned up all the resources associated with the parameter to the assignment operator. In other words, `clear` destroyed both the data we wanted to clean up and the data we were meaning to copy. The call to `copyOther` will therefore copy garbage data into the receiver object, since the resources it means to copy have already been cleaned up. This is extremely bad, and will almost certainly cause a program crash.

When writing assignment operators, you *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.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    if(this != &other) {
        clear();
        copyOther(other);
    }
}
```

Note that we check `if(this != &other)`. That is, we compare *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. In the practice problems for this chapter, you'll explore what would happen if you were to write `if(*this != other)`. One detail worth mentioning is that the self-assignment check is not necessary in the copy constructor, since an object can't be a parameter to its own 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;
```

This code is equivalent to `three = (two = one)`. Since our current assignment operator does not return a value, `(two = one)` does not have a value, so the above statement is meaningless and the code will not compile. We thus need to change our assignment operator so that performing an assignment like `two = one` yields a value that can then be assigned to other values. The final version of our assignment operator is thus

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

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. */ {
    /* Default initialization here. */
}

MyClass::MyClass(const MyClass& other) {
    copyOther(other);
}
```

```
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();
}
```

Semantic Equivalence and `copyOther` Strategies

Consider the following code snippet:

```
    Vector<int> one;
    Vector<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 the corresponding 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 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 values will be bitwise equivalent. Two objects are *semantically equivalent* if, like `one` and `two`, any operations performed on the objects will yield identical results. When writing a copy constructor and assignment operator, you attempt to convert an object into a semantically equivalent copy of another object. Consequently, 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 uses a shared 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 represents a mathematical set implemented as a 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);
    void clear();
};
```

This `ListSet` class exports two functions, `insert` and `contains`, that insert an element into the list and determine whether the list contains an element, respectively. This class represents a mathematical set, an *unordered* collection of elements, so the underlying linked list need not be in any particular order. For example, the lists `{0, 1, 2, 3, 4}` and `{4, 3, 2, 1, 0}` are semantically equivalent because checking whether a number is an element of the first list yields the same result as checking whether the number is in the second. In fact, any two lists containing the same elements are semantically equivalent to one another. This means that there are multiple ways in which we could implement `copyOther`. Consider these two:

```
/* Version 1: 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);
    }
}

/* Version 2: 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. This may initially seem problematic but is actually perfectly safe. 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 object and its copy.

There is one implementation of `copyOther` that is considerably more elegant than either of the two versions listed above:

```
/* Version 3: Duplicate list using the insert function */
template <typename T> void ListSet<T>::copyOther(const ListSet& other) {
    for(cellT* source = other.list; source != NULL; source = source->next)
        insert(source->data);
}
```

Notice that this implementation uses the `ListSet`'s public interface to insert the elements from the source `ListSet` into the receiver object. This version of `copyOther` is unquestionably the cleanest. If you'll notice, it doesn't matter exactly how `insert` adds elements into the list (indeed, `insert` could insert the elements at random positions), but we're guaranteed that at the end of the `copyOther` call, the receiver object will be semantically equivalent to the parameter.

Conversion Assignment Operators

When working with copy constructors, we needed to define an additional function, the assignment operator, to handle all the cases in which an object can be copied or assigned. However, in the chapter on conversion constructors, we provided a conversion constructor without a matching "conversion assignment operator." It turns out that this is not a problem because of how the assignment operator is invoked. Suppose that we have a `CString` class that has a defined copy constructor, assignment operator, and conversion constructor that converts raw C++ `char *` pointers into `CString` objects. Now, suppose we write the following code:

```
CString myCString;
myCString = "This is a C string!";
```

Here, in the second line, we assign an existing `CString` a new value equal to a raw C string. Despite the fact that we haven't defined a special assignment operator to handle this case, the above is perfectly legal code. When we write the line

```
myCString = "This is a C string!";
```

C++ converts it into the equivalent code

```
myCString.operator= ("This is a C string!");
```

This syntax may look entirely foreign, but is simply a direct call to the assignment operator. Recall that the assignment operator is a function named `operator =`, so this code passes the C string `"This is a C string!"` as a parameter to `operator =`. Because `operator =` accepts a `CString` object rather than a raw C string, C++ will invoke the `CString` conversion constructor to initialize the parameter to `operator =`. Thus this code is equivalent to

```
myCString.operator =(CString("This is a C string!"));
```

In other words, the conversion constructor converts the raw C string into a `CString` object, then the assignment operator sets the receiver object equal to this temporary `CString`.

In general, you need not provide a "conversion assignment operator" to pair with a conversion constructor. As long as you've provided well-defined copy behavior, C++ will link the conversion constructor and assignment operator together to perform the assignment.

Disabling Copying

In CS106B/X we provide you the `DISALLOW_COPYING` macro, which causes a compile error if you try to assign or copy objects of the specified type. `DISALLOW_COPYING`, however, is not a standard C++ feature. Without using the CS106B/X 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, instead we'll 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.

This trick is almost one hundred percent correct, but does have one edge case: what if we try to invoke the copy constructor or assignment operator inside a member function of the class? The copy functions might be private, but that doesn't mean that they don't exist, and if we call them inside a member function might accidentally create a copy of an otherwise uncopyable object. To prevent this from happening, we'll use a cute trick. Although we'll *prototype* the copy functions inside the private section of the class, we won't *implement* them. This means that if we accidentally do manage to call either function, we will get a linker error because the compiler can't find code for either function. This is admittedly a bit hackish, so in C++0x, the next revision of C++, there will be a way to explicitly indicate that a class is uncopyable. In the meantime, though, the above approach is perhaps your best option. We'll see another way to do this later when we cover inheritance.

Extended Example: `SmartPointer`

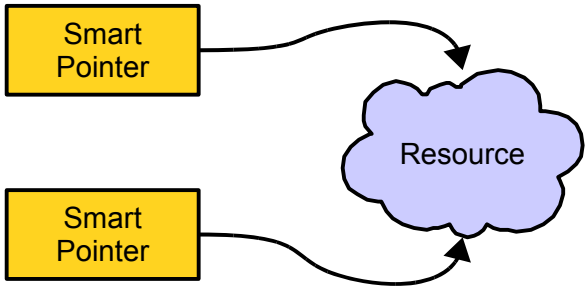
In C++ parlance, a raw pointer like an `int*` or a `char*` is sometimes called a *dumb pointer* because the pointer has no "knowledge" of the resource it owns. If an `int*` goes out of scope, it doesn't inform the object it's pointing at and makes no attempt whatsoever to clean it up. The `int*` doesn't own its resource, and assigning one `int*` to another doesn't make a deep copy of the resource or inform the other `int*` that another pointer now references its pointee.

Because raw pointers are so problematic, many C++ programmers prefer to use *smart pointers*, objects that mimic raw pointers but which perform functions beyond merely pointing at a resource. For example, the C++ standard library class `auto_ptr`, which we'll cover in the chapter on exception handling, acts like a regular pointer except that it automatically calls `delete` on the resource it owns when it goes out of scope. Other smart pointers are custom-tuned for specific applications and might perform functions like logging access, synchronizing multithreaded applications, or preventing accidental null pointer dereferences. Thanks to operator overloading, smart pointers can be built to look very similar to regular C++ pointers. We can provide an implementation of `operator *` to support dereferences like `*myPtr`, and can define `operator ->` to let clients write code to the effect of `myPtr->clear()`. Similarly, we can write copy constructors and assignment operators for smart pointers that do more than just transfer a resource.

Reference Counting

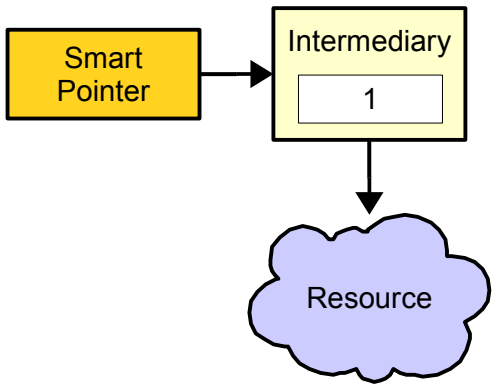
Memory management in C++ is tricky. You must be careful to balance every `new` with exactly one `delete`, and must make sure that no other pointers to the resource exist after `delete`-ing it to ensure that later on you don't access invalid memory. If you `delete` memory too many times you run into undefined behavior, and if you `delete` it too few you have a memory leak. Is there a better way to manage memory? In many cases, yes, and in this extended example we'll see one way to accomplish this using a technique called *reference counting*. In particular, we'll design a smart pointer class called `SmartPointer` which acts like a regular C++ pointer, except that it uses reference counting to prevent resource leaks.

To motivate reference counting, let's suppose that we have a smart pointer class that stores a pointer to a resource. The destructor for this smart pointer class can then `delete` the resource automatically, so clients of the smart pointer never need to explicitly clean up any resources. This system is fine in restricted circumstances, but runs into trouble as soon as we have several smart pointers pointing to the same resource. Consider the scenario below:



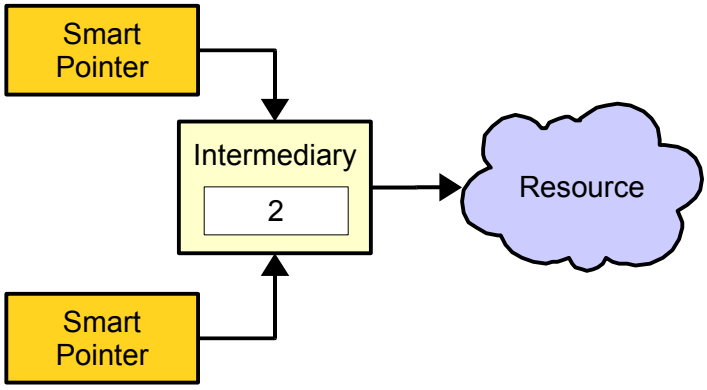
Both of these pointers can access the stored resource, but unfortunately neither smart pointer knows of the other's existence. Here we hit a snag. If one smart pointer cleans up the resource while the other still points to it, then the other smart pointer will point to invalid memory. If both of the pointers try to reclaim the dynamically-allocated memory, we will encounter a runtime error from double-`delete`-ing a resource. Finally, if neither pointer tries to clean up the memory, we'll get a memory leak.

To resolve this problem, we'll use a system called *reference counting* where we will explicitly keep track of the number of pointers to a dynamically-allocated resource. While there are several ways to make such a system work, perhaps the simplest is to use an intermediary object. This can be seen visually:



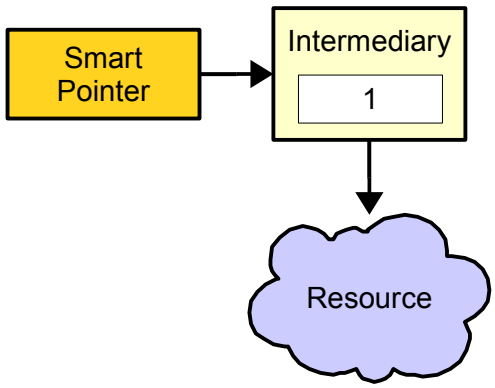
Now, the smart pointer stores a pointer to an intermediary object rather than a pointer directly to the resource. This intermediary object has a counter (called a *reference counter*) that tracks the number of smart pointers accessing the resource, as well as a pointer to the managed resource. This intermediary object lets the smart pointers tell whether or not they are the only pointer to the stored resource; if the reference count is anything other than one, some other pointer shares the resource. Provided that we accurately track the reference count, each pointer can tell if it's the last pointer that knows about the resource and can determine whether to deallocate it.

To see how reference counting works, let's walk through an example. Given the above system, suppose that we want to share the resource with another smart pointer. We simply make this new smart pointer point to the same intermediary object as our original pointer, then update the reference count. The resulting scenario looks like this:

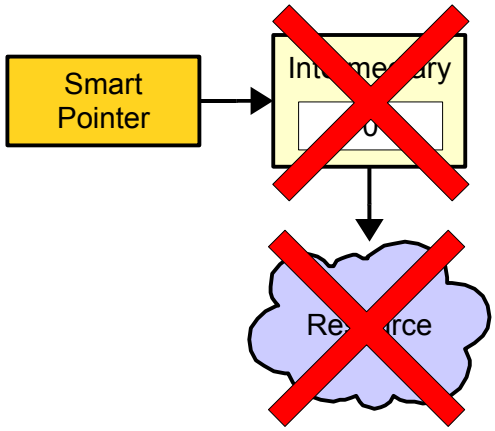


Although in this diagram we only have two objects pointing to the intermediary, the reference-counting system allows for any number of smart pointers to share a single resource.

Now, suppose one of these smart pointers needs to stop pointing to the resource – maybe it's being assigned to a different resource, or perhaps it's going out of scope. That pointer decrements the reference count of the intermediary variable and notices that the reference count is nonzero. This means that at least one smart pointer still references the resource, so the smart pointer simply leaves the resource as it is. Memory now looks like this:



Finally, suppose this last smart pointer needs to stop pointing to this resource. It decrements the reference count, but this time notices that the reference count is zero. This means that no other smart pointers reference this resource, and the smart pointer knows that it needs to deallocate the resource and the intermediary object, as shown here:



The resource has now been deallocated and no other pointers reference the memory. We've safely and effectively cleaned up our resources. Moreover, this process is completely automatic – the user never needs to explicitly deallocate any memory.

The following summarizes the reference-counting scheme described above:

- When creating a smart pointer to manage newly-allocated memory, first create an intermediary object and make the intermediary point to the resource. Then, attach the smart pointer to the intermediary and set the reference count to one.

- To make a new smart pointer point to the same resource as an existing one, make the new smart pointer point to the old smart pointer's intermediary object and increment the intermediary's reference count.

- To remove a smart pointer from a resource (either because the pointer goes out of scope or because it's being reassigned), decrement the intermediary object's reference count. If the count reaches zero, deallocate the resource and the intermediary object.

While reference counting is an excellent system for managing memory automatically, it does have its limitations. In particular, reference counting can sometimes fail to clean up memory in "reference cycles," situations where multiple reference-counted pointers hold references to one another. If this happens, none of the reference counters can ever drop to zero, since the cyclically-linked elements always refer to one another. But barring this sort of setup, reference counting is an excellent way to automatically manage memory. In this extended example, we'll see how to implement a reference-counted pointer, which we'll

call `SmartPointer`, and will explore how the correct cocktail of C++ constructs can make the resulting class slick and efficient.

Designing `SmartPointer`

The above section details the *implementation* the `SmartPointer` class, but we have not talked about its *interface*. What functions should we provide? We'll try to make `SmartPointer` resemble a raw C++ pointer as closely as possible, meaning that it should support `operator *` and `operator ->` so that the client can dereference the `SmartPointer`. Here is one possible interface for the `SmartPointer` class:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

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

Here is a breakdown of what each of these functions should do:

`explicit SmartPointer(T* memory);`

Constructs a new `SmartPointer` that manages the resource specified as the parameter. The reference count is initially set to one. We will assume that the provided pointer came from a call to `new`. This function is marked `explicit` so that we cannot accidentally convert a regular C++ pointer to a `SmartPointer`. At first this might seem like a strange design decision, but it prevents a wide range of subtle bugs. For example, suppose that this constructor is not `explicit` and consider the following function:

```
void PrintString(const SmartPointer<string>& ptr) {
    cout << *ptr << endl;
}
```

This function accepts a `SmartPointer` by reference-to-`const`, then prints out the stored string. Now, what happens if we write the following code?

```
string* ptr = new string("Yay!");
PrintString(ptr);
delete ptr;
```

The first line dynamically-allocates a `string`, passes it to `PrintString`, and finally deallocates it. Unfortunately, this code will almost certainly cause a runtime crash. The problem is that `PrintString` expects a `SmartPointer<string>` as a parameter, but we've provided a `string*`. C++ notices that the `SmartPointer<string>` has a conversion constructor that accepts a `string*`, and makes a temporary `SmartPointer<string>` using the pointer we passed as a parameter. This new `SmartPointer` starts tracking the pointer with a reference count of one. After the function returns, the parameter is cleaned up and its destructor invokes. This decrements the reference count to zero, and then deallocates the pointer stored in the `SmartPointer`. The above code then tries to `delete ptr` a second time, causing a runtime crash. To prevent this problem, we'll mark the constructor `explicit`, which makes the implicit conversion illegal and prevents this buggy code from compiling.

`SmartPointer(const SmartPointer& other);` Constructs a new `SmartPointer` that shares the resource contained in another `SmartPointer`, updating the reference count appropriately.
`SmartPointer& operator=(const SmartPointer& other);` Causes this `SmartPointer` to stop pointing to the resource it's currently managing and to share the resource held by another `SmartPointer`. If the smart pointer was the last pointer to its resource, it `delete`s it.
`~SmartPointer();` Detaches the `SmartPointer` from the resource it's sharing, freeing the associated memory if necessary.
`T& operator* () const;` "Dereferences" the pointer and returns a reference to the object being pointed at. Note that `operator*` is `const`; see the last chapter for more information why.
`T* operator-> () const;` Returns the object that the arrow operator should really be applied to if the arrow is used on the `SmartPointer`. Again, see the last chapter for more information on this.

Given this public interface for `SmartPointer`, we can now begin implementing the class. We first need to decide on how we should represent the reference-counting information. One simple method is to define a private `struct` inside `SmartPointer` that represents the reference-counting intermediary. This looks as follows:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

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

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;
};
```

Here, the `resource` field of the `Intermediary` is the actual pointer to the stored resource and `refCount` is the reference count. Notice that we did not declare the reference count as a direct data member of the `SmartPointer`, but rather in the `Intermediary` object. This is because the reference count of a resource is not owned by any one `SmartPointer`, but rather is shared across all `SmartPointer`s that point to a particular resource. This way, any changes to the reference count by one `SmartPointer` will become visible in all of the other `SmartPointer`s referencing the resource. You might ask – could we have made the `refCount` a `static` data member? This would indeed make the reference count visible across multiple `SmartPointer`s, but unfortunately it won't work out correctly. In particular, if we use `SmartPointer` to

manage multiple resources, each one needs to have its own `refCount` or changes to the `refCount` for a particular *resource* will show up in the `refCount` for other resources.

Given this setup, we can implement the `SmartPointer` constructor by creating a new `Intermediary` that points to the specified resource and has an initial reference count of one:

```
template <typename T> SmartPointer<T>::SmartPointer(T* res) {
    data = new Intermediary;
    data->resource = res;
    data->refCount = 1;
}
```

It's very important that we allocate the `Intermediary` object on the heap rather than as a data member. That way, when the `SmartPointer` is cleaned up (either by going out of scope or by an explicit call to `delete`), if it isn't the last pointer to the shared resource, the intermediary object isn't cleaned up.

We can similarly implement the destructor by decrementing the reference count, then cleaning up memory if appropriate. Note that if the reference count hits zero, we need to delete both the resource *and* the intermediary. Forgetting to deallocate either of these leads to memory leaks, the exact problem we wanted to avoid. The code for this is shown here:

```
template <typename T> SmartPointer<T>::~SmartPointer() {
    --data->refCount;
    if(data->refCount == 0) {
        delete data->resource;
        delete data;
    }
}
```

This is an interesting destructor in that it isn't guaranteed to actually clean up any memory. Of course, this is exactly the behavior we want, since the memory might be shared among multiple `SmartPointer`s.

Implementing `operator *` and `operator ->` simply requires us to access the pointer stored inside the `SmartPointer`. These two functions can be implemented as follows:*

```
template <typename T> T& SmartPointer<T>::operator * () const {
    return *data->resource;
}
template <typename T> T* SmartPointer<T>::operator -> () const {
    return data->resource;
}
```

Now, we need to implement the copy behavior for this `SmartPointer`. One way to do this is to write helper functions clear and `copyOther` which perform `deallocation` and copying. We will use a similar approach here, except using functions named `detach` and `attach` to make explicit the operations we're performing. This leads to the following definition of `SmartPointer`:

---

* It is common to see `operator ->` implemented as

```
RetType* MyClass::operator -> () const
{
    return &**this;
}
```

`&**this` is interpreted by the compiler as `&(*(*this))`, which means "dereference the `this` pointer to get the receiver object, then dereference the receiver. Finally, return the address of the referenced object." At times this may be the best way to implement `operator ->`, but I advise against it in general because it's fairly cryptic.

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

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

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};
```

Now, what should these functions do? The first of these, `detach`, should detach the `SmartPointer` from the shared intermediary and clean up the memory if it was the last pointer to the shared resource. In case this sounds familiar, it's because this is exactly the behavior of the `SmartPointer` destructor. To avoid code duplication, we'll move the code from the destructor into detach as shown here:

```
template <typename T> void SmartPointer<T>::detach() {
    --data->refCount;
    if(data->refCount == 0) {
        delete data->resource;
        delete data;
    }
}
```

We can then implement the destructor as a wrapped call to detach, as seen here:

```
template <typename T> SmartPointer<T>::~SmartPointer() {
    detach();
}
```

The attach function, on the other hand, makes this SmartPointer begin pointing to the specified Intermediary and increments the reference count. Here's one possible implementation of attach:

```
template <typename T> void SmartPointer<T>::attach(Intermediary* to) {
    data = to;
    ++data->refCount;
}
```

Given these two functions, we can implement the copy constructor and assignment operator for `SmartPointer` as follows:

```
template <typename T> SmartPointer<T>::SmartPointer(const SmartPointer& other){
    attach(other.data);
}

template <typename T>
SmartPointer<T>& SmartPointer<T>::operator= (const SmartPointer& other) {
    if(this != &other) {
        detach();
        attach(other.data);
    }
    return *this;
}
```

It is crucial that we check for self-assignment inside the `operator=` function, since otherwise we might destroy the data that we're trying to keep track of!

At this point we have a rather slick `SmartPointer` class. Here's some code demonstrating how a client might use `SmartPointer`:

```
SmartPointer<string> myPtr(new string);
*myPtr = "This is a string!";
cout << *myPtr << endl;

SmartPointer<string> other = myPtr;
cout << *other << endl;
cout << other->length() << endl;
```

The beauty of this code is that client code using a `SmartPointer<string>` looks almost identical to code using a regular C++ pointer. Isn't operator overloading wonderful?

Extending `SmartPointer`

The `SmartPointer` defined above is useful but lacks some important functionality. For example, suppose that we have the following function:

```
void DoSomething(string* ptr);
```

Suppose that we have a `SmartPointer<string>` managing a resource and that we want to pass the stored string as a parameter to `DoSomething`. Despite the fact that `SmartPointer<string>` mimics a `string*`, it technically is not a `string*` and C++ won't allow us to pass the `SmartPointer` into `DoSomething`. Somehow we need a way to have the `SmartPointer` hand back the resource it manages.

Notice that the only `SmartPointer` member functions that give back a pointer or reference to the actual resource are `operator*` and `operator->`. Technically speaking, we *could* use these functions to pass the stored `string` into `DoSomething`, but the syntax would be messy (in the case of `operator*`) or nightmarish (for `operator ->`). For example:

```
SmartPointer<string> myPtr(new string);

/* To use operator* to get the stored resource, we have to first dereference
 * the SmartPointer, then use the address-of operator to convert the returned
 * reference into a pointer.
 */
DoSomething(&*myPtr);

/* To use operator-> to get the stored resource, we have to explicitly call the
 * operator-> function.  Yikes!
 */
DoSomething(myPtr.operator-> ());
```

Something is clearly amiss and we cannot reasonably expect clients to write code like this routinely. We'll need to extend the `SmartPointer` class to provide a way to return the stored pointer directly. This necessitates the creation of a new member function, which we'll call `get`, to do just that. Given a function like this, we could then invoke `DoSomething` as follows:

```
DoSomething(myPtr.get());
```

The updated interface for `SmartPointer` looks like this:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

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

    T* get() const;

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};
```

The implementation of `get` is fairly straightforward and is shown here:

```
template <typename T> T* SmartPointer<T>::get() const {
    return data->resource;
}
```

Further Extensions

There are several more extensions to the `SmartPointer` class that we might want to consider, of which this section explores two. The first is rather straightforward. At times, we might want to know exactly how many `SmartPointer`s share a resource. This might enable us to perform some optimizations, in par-

ticular a technique called *copy-on-write*. We will not explore this technique here, though you are encouraged to do so on your own.

Using the same logic as above, we'll define another member function called `getShareCount` which returns the number of `SmartPointer`s pointing to the managed resource (including the receiver object). This results in the following class definition:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

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

    T*  get() const;
    size_t getShareCount() const;

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};
```

And the following implementation:

```
template <typename T> size_t SmartPointer<T>::getShareCount() const {
    return data->refCount;
}
```

The last piece of functionality we'll consider is the ability to "reset" the `SmartPointer` to point to a different resource. When working with a `SmartPointer`, at times we may just want to drop whatever resource we're holding and begin managing a new one. As you might have suspected, we'll add yet another member function called `reset` which resets the `SmartPointer` to point to a new resource. The final interface and code for `reset` is shown here:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

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

    T*     get() const;
    size_t getShareCount() const;
    void   reset(T* newRes);

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

template <typename T> void SmartPointer<T>::reset(T* newRes) {
    /* We're no longer associated with our current resource, so drop it. */
    detach();

    /* Attach to a new intermediary object. */
    data = new Intermediary;
    data->resource = newRes;
    data->refCount = 1
}
```

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. When is the copy constructor invoked?
2. When is the assignment operator invoked?
3. What is the signature of the copy constructor?
4. What is the signature of the assignment operator?
5. What is the rule of three? What are the "three" it refers to?
6. What is the behavior of the default-generated copy constructor and assignment operator?
7. Why does the assignment operator have to check for self-assignment but the copy constructor not need to check for "self-initialization?"
8. What is bitwise equivalence? What is semantic equivalence? Which of the two properties should be guaranteed by the two copy functions?

9. What is a smart pointer?

10. What is reference-counting?

11. Realizing that the copy constructor and assignment operator for most classes have several commonalities, you decide to implement a class's copy constructor using the class's assignment operator. For example, you try implementing the `Vector`'s copy constructor as

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    *this = other;
}
```

(Since `this` is a pointer to the receiver object, `*this` is the receiver object, so `*this = other` means to assign the receiver object the value of the parameter `other`)

This idea, while well-intentioned, has a serious flaw that causes the copy constructor to almost always cause a crash. Why is this? *(Hint: Were any of the `Vector` data members initialized before calling the assignment operator? Walk through the assignment operator and see what happens if the receiver object's data members haven't been initialized.)*

12. It is illegal to write a copy constructor that accepts its parameter by value. Why is this? However, it's perfectly acceptable to have an assignment operator that accepts its parameter by value. Why is this legal? Why the difference?

13. An alternative implementation of the assignment operator uses a technique called *copy-and-swap*. The copy-and-swap approach is broken down into two steps. First, we write a member function that accepts a reference to another instance of the class, then exchanges the data members of the receiver object and the parameter. For example, when working with the `DebugVector`, we might write a function called `swapWith` as follows:

```
template <typename ElemType> void Vector<ElemType>::swapWith(Vector& other)
{
    swap(array, other.array);
    swap(logicalLength, other.logicalLength);
    swap(allocatedLength, other.allocatedLength);
}
```

Here, we use the STL `swap` algorithm to exchange data members. Notice that we never actually make a deep-copy of any of the elements in the array – we simply swap pointers with the other `DebugVector`. We can then implement the assignment operator as follows:

```
template <typename T> Vector<T>& Vector<T>::operator= (const Vector& other)
{
    DebugVector temp(other);
    swapWith(temp);
    return *this;
}
```

Trace through this implementation of the assignment operator and explain how it sets the receiver object to be a deep-copy of the parameter. What function actually deep-copies the data? What function is responsible for cleaning up the old data members?

14. When writing an assignment operator using the pattern covered earlier in the chapter, we had to explicitly check for self-assignment in the body of the assignment operator. Explain why this is no longer necessary using the copy-and-swap approach, but why it still might be a good idea to insert the self-assignment check anyway.

15. A singleton class is a class that can have at most one instance. Typically, a singleton class has its default constructor and destructor marked private so that clients cannot instantiate the class directly, and exports a `static` member function called `getInstance()` that returns a reference to the only instance of the class. That one instance is typically a private static data member of the class. For example:

```
class Singleton {
public:
    static Singleton& getInstance();

private:
    Singleton();  // Clients cannot call this function; it's private
    ~Singleton(); // ... nor can they call this one

    static Singleton instance; // ... but they can be used here because
                               // instance is part of the class.
};

Singleton Singleton::instance;
```

Does it make sense for a singleton class to have a copy constructor or assignment operator? If so, implement them. If not, modify the `Singleton` interface so that they are disabled.

16. Given this chapter's description about how to disable copying in a class, implement a macro `DISALLOW_COPYING` that accepts as a parameter the name of the current class such that if `DISALLOW_COPYING` is placed into the private section of a class, that class is uncopyable. Note that it is legal to create macros that span multiple lines by ending each line with the `\` character. For example, the following is all one macro:

```
#define CREATE_PRINTER(str) void Print##str() {\
    cout << #str << endl;\
}
```

17. Consider the following alternative mechanism for disabling copying in a class: instead of marking those functions private, instead we implement those functions, but have them call `abort` (a function from `<cstdlib>` that immediately terminates the program) after printing out an error message. For example:

```
class PseudoUncopyable {
 public:
    PseudoUncopyable(const PseudoUncopyable& other) {
        abort();
    }
    PseudoUncopyable& operator= (const PseudoUncopyable& other) {
        abort();
        return *this; // Never reached; suppresses compiler warnings
    }
};
```

Why is this approach a bad idea?

18. Should you copy `static` data members in a copy constructor or assignment operator? Why or why not?

19. In the canonical implementation of the assignment operator we saw earlier in this chapter, we used the check `if (this != &other)` to avoid problems with self-assignment. In this exercise, we'll see what happens if we replace this check with `if (*this != other)`.

    1. What is the meaning of `if (*this != other)`? Will this code compile for any class, or does that class have to have a special property?

    2. Will the check `if (*this != other)` correctly detect whether an object is being assigned to itself? Will it detect anything else?

    3. Assume that the `Vector` has an implementation of `operator!=` that checks whether the operands have exactly the same size and elements. What is the asymptotic (big-O) complexity of the check `if(*this != other)`? How about `if (this != &other)`? Does this give you a better sense why the latter is preferable to the former?

20. In a sense, our implementation of the `Vector` assignment operator is wasteful. It works by completely discarding the internal array, then constructing a new array to hold the other `Vector`'s elements. An alternative implementation would work as follows. If the other `Vector`'s elements can fit in the space currently allocated by the `Vector`, then the elements from the other `Vector` are copied directly into the existing space. Otherwise, new space is allocated as before. Rewrite the `Vector`'s `operator=` function using this optimization. Why won't this technique work for the copy constructor?