CS225 : Data Structures and Software Engineering
Constructors and the Big Three

Jason Zych

Chapter 1

Constructors and the Big Three

1.1 The default constructor

The constructor is a function that specifies how an object should be initialized upon creation. A constructor has the same name as the corresponding class, but never has a return type. For example, for class `Coord`, the constructor would be called `Coord()` as well. The constructor can have parameters, and it can also be overloaded, meaning you can have multiple constructors, all with the same name, just with differing parameter lists. Constructors can initialize values within the object that you are creating and perform other "object start-up" tasks as well.

A constructor without any parameters is called a default constructor. Below is an example of a default constructor for the `Coord` class. (Note that we are putting the definition of the functions directly into the class declaration here, rather than writing them in a separate implementation file as we discussed in lecture 1. Technically, it *is* okay to do this; however, it is not generally the way to do things, for reasons we're not going to discuss quite yet. That is why we told you to separate the declaration and definition in lecture 1. The only reason we are not doing so here is to save space on the paper; if we were providing example code files, we *would* separate the two in those code files, and you should do so as well.)

```
class Coord {
   public:
      Coord() {xCoord = 0.0; yCoord = 0.0;}
      void Initialize(double x, double y) {xCoord = x; yCoord = y;}
      void Print() { cout << "(" << xCoord << "," << yCoord << ")" << endl;}
   private:
     double xCoord, yCoord;
};
```

To use the constructor, you could have a statement such as:

```
Coord* cPtr = new Coord();
```

The command `new Coord()` allocates space for a new `Coord` object in memory, as we have seen. But, since `Coord()` is a constructor, this command doesn't just allocate space – it also runs the code in the constructor and initializes `xCoord` and `yCoord` to `0.0`. Recall that back in the pointer packet we mentioned that one of the tasks of the `new` command was to make

the appropriate constructor call. Above, you now see what was meant by that. When you dynamically allocate an object using `new`, you first have the memory set aside for the object, *then* the constructor is called to initialize that memory, and finally, the address of the dynamic object is returned by the `new` command.

You could also have said:

```
Coord c1, c2;
```

and for each of the objects `c1` and `c2`, the memory is set aside on the function stack for the object, and then the constructor gets called and both member data of the object are initialized to `0.0`. (Note that there are not parenthesis after `c1` or `c2`; we will discuss why shortly.)

The default constructor is generally used to initialize the object to the "default state" or "start state". Usually, this means that you want to have variables set to *something*, or else you could have garbage in their memory locations, but you also don't currently have anything specific you want stored there. So, integers generally get set to `0`, pointers get set to `NULL`, and so on. But that is just the common situation; certainly, your own specific class might have better values to set the variables to by default, and if so, then those values are what you should use. It all depends on what is appropriate for your specific class.

1.2 Overloading functions

Function overloading (or *method overloading*, as it was called in Java) is the term for defining more than one function with the same name. If you think of a function prototype:

```
void foo(int a, int b);
```

the function is uniquely defined by the name and parameter list. Certainly, you have seen functions with different names but the same parameter list. For example, you might pass an integer to a function that squares its parameter (ex. `int square(int x)`), or pass an integer to a function that returns the corresponding fibonacci number (ex. `int fib(int x)`), or pass an integer to a function that draws a circle with that radius (ex. `void DrawCircle(int x)`). Those would be three different functions that each would have a single integer as a parameter; they would be distinguished by the fact that they all had different names.

Similarly, two functions can have the same name, as long as the parameter lists are different. In that way, even though the functions can't be distinguished by name, they can be distinguished by parameter list. If two functions had the same name *and* the same parameter list, then it would never be clear which one you were trying to call, and so the compiler would indicate that you have an error. But as long as either the name *or* the parameter list is different, you are okay.

Why on earth would you want two functions with the same name? Well, imagine you have written a function called `add`, that adds two integers:

```
int add(int a, int b)
{
   return (a+b);
}
```

Now, you may find in your code that there is a use for a function that adds four integers as well. Adding four integers together using just the above function would take three calls to the function. It would be easier if there was a function available to do it in one. But what should we call it? One good name is `add`, but `add` is already taken.

The beauty of overloading is that it doesn't matter that `add` is already taken. We can use it again anyway, as long as we have a different parameter list :

```
int add(int a, int b, int c, int d)
{
   return (a+b+c+d);
}
```

When your program gets compiled, and you have a statement somewhere like:

```
add(5,6,7,8);
```

the compiler sees that the argument list has four integers, and tries to find a function called `add` with a parameter list of four integers. It will first see the `add` with two integers, and will decide that that is not the function it wants. It will then see the `add` with four integers in the parameter list, and realize that that is the function it does want. And therefore compilation works correctly. If you had not written the `add` with four parameters, and then tried to call it, you would of course get a compiler error.

The key idea is ambiguity – or rather, avoiding it. When the parameter lists of two functions are different, there's no ambiguity about which function you are trying to call, so if the names happen to be the same, that's fine.

1.3 Overloading the constructor

So, now consider the situation where we create a `Coord` object, but where we know exactly what values we want the `Coord` to hold, even before we create the object itself.

If you want to be able to set the data to actual user-chosen values to start with, you can overload the constructor with a function that takes arguments:

```
class Coord {
   public:
      Coord() {xCoord = 0.0; yCoord = 0.0;}
      Coord(double x, double y) {xCoord = x; yCoord = y;}
      void Initialize(double x, double y) {xCoord = x; yCoord = y;}
      void Print() { cout << "(" << xCoord << "," << yCoord << ")" << endl;}
   private:
     double xCoord, yCoord;
};
```

Note the second constructor (with a different parameter list!) in the class now. You could use this constructor to initialize local objects as follows:

```
Coord firstVal(0.0, 4,5);
Coord secondVal(5.3, 3.2);
```

or to initialize dynanmic objects as follows:

```
Coord* cPtr = new Coord(7.2, 5.2);
```

When the constructor is called in any of the three lines above, the member data of the object gets initialized to the user-chosen values (the arguments to the above constructor calls), rather than zero as is the case with the default constructor. The fact that we had two arguments of type `double` rather than no arguments at all made it clear we wanted the second constructor rather than the first. In general, you could have many different constructors for a class; the constructor you want to use is always clear from the arguments you are using on the declaration or allocation line.

However, there is one small syntax quirk you need to be familiar with. Note that for both the default constructor and the constructor with parameters, you have parenthesis after "`new Coord`" but with the default constructor you only have parenthesis when you are allocating the object dynamically and not when you are allocating the object locally. (We pointed this out above, remember?) The rule for almost every constructor call – whether for a dynamic or local object – is that you'll have parenthesis, possibly empty, which contain the arguments you are trying to send into the constructor. In the special case of a local object and the default constructor, however, you do *not* use parenthesis. That is, your declaration looks like this:

```
Coord  c4;
```

and not like this:

```
Coord c4();     // note the parenthesis -- incorrect!
```

Why is this? Why don't we have parenthesis in the special case of local declaration and default constructor even though we have them for all the other cases? Well, the reason is that if we were to use the parenthesis, the compiler could also view this as the declaration of a function called `c4` that takes no parameters and has a return type of `Coord`. We do not have this problem when using "`new`", since lines using "`new`" clearly are not function declarations. We also don't have this problem when declarating local variables using constructors with parameters. This is because function parameter lists require types to appear on the parameter lists, so if you write

```
Coord foo(int x, int y);
```

then you are dealing with a function declaration for a function called "`foo`", but the line of code

```
Coord foo(0.0, 4.5);
```

has no types involved – there are arguments instead of type/variable pairs – and thus is clearly not a function declaration but rather is a function call (specifically, a constructor call), and thus a variable declaration for `foo`. As long as you have even one item in the parenthesis, then you can tell whether your line of code is a variable declaration or a function declaration based on whether that item in the parenthesis is a type/variable pair, or a function argument of some kind. But if the parenthesis are empty, you don't know if they are empty because there are no parameters, or if they are empty instead because there are no arguments. Thus, it is unclear whether it is a function declaration, or if it is instead a variable declaration using a default constructor call.

The language C++ needed to be compatible with code written in the language C, and since it was already established in C that the syntax:

```
Coord c4();
```

was the form for a function declaration, when C++ was designed, that was kept as the form for a function declaration, and local variable declarations using the default constructor became a special case where the parenthesis were left off, so as not to confuse such a declaration with a function declaration. So, the case of using the default constructor to initialize a local variable is a special case – one where the empty constructor parenthesis are understood to be there, but you don't actually type them in so as not to confuse the compiler and make it think you are attempting a function declaration. If you *do* type those parenthesis anyway, the confusion the compiler will face is similar to the confusion you will cause by forgetting your semicolon at the end of the class declaration. So, if you do happen to get some really strange errors, this is another one of those odd errors that might be the cause and something you should check your code for (just like you should check your code to see if you forgot the semicolon at the end of a class).

1.4 Default constructor issues

Part of the reason the no-argument constructor is also called the default constructor is that you are sometimes given this constructor by default. If you do not explicitly write any constructors of your own into your code for a particular class, then the compiler will give you the no-argument constructor automatically. It won't magically appear in your source code, of course – but the compiler will write it into the compiled code. The definition of this constructor will be empty, as if you had done this yourself:

```
Coord::Coord()
{
     // no code here
}
```

So, nothing will actually be *done* by the default constructor in this case; it's only included by the compiler so that there is some constructor to call. Every variable declaration and object allocation of a user-defined type will call a constructor somehow, so if there was no constructor for a class, the class would be useless since you could never create any objects of that class. Therefore, the compiler at least provides you with this constructor if you have not provided any of your own.

However, as soon as you write any constructor, the compiler places all responsibility on you and does not provide a default constructor. Therefore, you can't write only

```
Coord(double x, double y);
```

into your class and hope to also have

```
Coord();
```

available to you. If you want the default constructor available to you, you either need to write it in yourself (just like you need to write in any other constructor if you need it), or else write in *no* constructors, so that the compiler will give you the default constructor automatically. (Note this is the same rule that you had to deal with in Java, so it shouldn't be something entirely new.)

The default constructor is used very frequently. Part of the reason it is called the "default" constructor is that it is called by default in various places. For example, if you create an array of six `Coord` objects:

```
Coord myArray[6];
```

then each of those six objects is initialized using the default constructor. You don't have any way to select the use of a different constructor there, and if you don't have a default constructor available, the array declaration won't work (i.e. won't compile – since the rule is to use a default constructor and the compiler can't find one). Because of "hidden" uses of the default constructor like that, it is usually a very good idea for a class to have a default constructor – and therefore if you write any constructor at all for a class (thus preventing the compiler from giving you a default constructor automatically), you should make sure to write a default constructor for the class (thus ensuring that the class has a default constructor even though the compiler didn't provide it one by default). You can always later overwrite any values assigned by the default constructor (as we shall soon see), but at the very least, you want the default constructor available to provide the *first* assignment of values, in all the places where that default constructor is the choice that the language either chooses for you by default, or worse, forces upon you (as in the array example above).

1.5 The copy constructor

There are many different constructors you could write for a class, because there are many different parameter lists you might find useful for initializing your object. A constructor that takes as a parameter another object of the same class is called a *copy constructor*, because you are initializing your object to be a copy of the object you passed in as a parameter.

```
class Coord {
   public:
      Coord() {xCoord = 0.0; yCoord = 0.0;}
      Coord(const Coord& origVal);
      void Initialize(double x, double y) {xCoord = x; yCoord = y;}
      void Print() { cout << "(" << xCoord << "," << yCoord << ")" << endl;}
   private:
     double xCoord, yCoord;
};
```

The second constructor in the code above is our copy constructor. (We've left out the definition code and provided only the declaration, as we are really supposed to do in the first place. We'll provide the definition in a moment.) All we have done here is overloaded the constructor, just as we did before. The only thing that has changed is the specific parameter list. The second line of code below uses this constructor.

```
Coord c1;       // uses default constructor
Coord c2(c1);   // uses copy constructor; c2 is initialized
                //   to be a copy of c1
```

Note the actual definition code of the copy constructor, which appears below:

```
Coord::Coord(const Coord& origVal)
{
   xCoord = origVal.xCoord;
   yCoord = origVal.yCoord;
}
```

The fact that `xCoord` and `yCoord` are private variables does not prevent us from writing code like the code above. Just as in Java, when a member variable of a class is `private`, that only means it cannot be directly accessed outside of its class. However, *within* its class – i.e. within the member functions of its class – it can be used freely. So, above, we not only have access to the `xCoord` and `yCoord` of the new object we are initializing; we also have access to the `xCoord` and `yCoord` of the object we passed in as an argument (i.e. the object called "`origVal`"). Within the member functions of the class `Coord`, we have access to any `private` variables of any `Coord` object. And in general, for any class, within the member functions of that class we have access to any `private` variables of any object of that class. This is intentional – if we did not, we could not write things like a copy constructor because we could never read the variables of some other identical-type object in order to copy them.

In addition to being used explicitly as in the code above, there are also some "hidden uses" for the copy constructor – times when the compiler puts a copy constructor call into the compiled code without you being aware of it. (We will discuss some of these situations in just a moment.) For that reason, the copy constructor is very important, and therefore the compiler will *always* provide a copy constructor if you don't write one. This is not like the default constructor, where *sometimes* you get one from the compiler, under certain conditions. Quite the contrary – in the case of the copy constructor, you will *always* be given one by default if you don't write one yourself. This means that every class will have a copy constructor available, since the only way to keep the compiler from giving you one for that class is to write one yourself, and in that case, the class still has a copy constructor.

If the copy constructor is given to you automatically by the compiler, what does it do? Well, by default, the copy constructor performs what is known as a *memberwise* copy – that is, values are copied data member by data member. If the data member is a primitive type, then it's just a simple pattern of bits, and the compiler will just copy that bit pattern. This is known as a *bitwise copy*. If the data member is of a user-defined type, then the compiler assumes the copy constructor for that user-defined type is correctly written and will correctly copy the object, and uses that copy constructor to copy the user-defined type.

For example, if we had a class `String` that was designed to store text strings, then imagine if this were the data member section of our `Coord` class:

```
class Coord {
   public:
         // whatever you want here
   private:
     double xCoord, yCoord;
     String s;
};
```

In this case, the default version of the copy constructor (i.e. the one given to you by the compiler) will perform bitwise copies of the `xCoord` and `yCoord` variables, since they are of type `double`,

a primitive type. So, the `xCoord` for your new object will just hold the same bit pattern – and thus the same value – as the `xCoord` of the object you are copying from. As far as the `String` variable `s` goes, well, we have no idea what kind of data members are used to implememt a `String` object, since those would be `private` and we can't look at the `private` data of some other class. However, the compiler assumes that whatever is going on in the `String` class, the copy constructor for the `String` class correctly handles it. So, the variable `s` in the new object will be initialized by calling the `String` copy constructor, and passing in as an argument to that `String` copy constructor the variable `s` of the `Coord` object you are copying from. Thus, when all of that is done, we have accurately copied `xCoord`, `yCoord`, and `s` from one `Coord` object – the one that was passed as an argument to the copy constructor – and into the other `Coord` object – the one which the copy constructor was actually invoked to initialize.

If the compiler-provided version does all that for us, why would we ever want to actually write a copy constructor of our own? The code we wrote earlier:

```
Coord::Coord(const Coord& origVal)
{
   xCoord = origVal.xCoord;
   yCoord = origVal.yCoord;
}
```

is exactly what the default version of the copy constructor would have done, so in the above case, we didn't need to write that code. We could have just relied on the compiler-supplied default, which would have done exactly what we did ourselves. And in fact, many times we will indeed be able to just rely on the default version of the copy constructor and not bother with it writing it ourselves. There *are* situations where we need to supply our own copy constructor – situations where the default behavior is not what we want – but we will first discuss destructors, and then we will return to copy constructors and discuss the situations where the default behavior of the copy constructor is not sufficient for our needs.

For the moment, though, we have one more thing to discuss before we move on. We had said that there were times when the compiler used the copy constructor even though we hadn't explicitly written a call to that constructor. One such example is when we pass an object using pass-by-value. Remember that when we use pass-by-value, the system makes a copy of the argument we pass in, and operates on that copy. Well, as our objects get more complex, it will not necessarily be automatically clear to the system how that copy should be made. Hence, the copy constructor is used. Whenever you pass an argument of a user-defined type by value, the way the system manages to make a correct copy of that argument is by calling the copy constructor and letting the copy constructor correctly initialize the copy, using the original argument as the parameter it reads the information from. Similarly, if you return by value (rather than by reference), the copy constructor is also used to make that copy.

This is why you are required to pass your argument to the copy constructor by reference, and not by value. Remember that this was our declaration for the copy constructor:

```
Coord(const Coord& origVal);
```

We can certainly require the address of `origVal` – that is, pass a `Coord*` – if we want to, but such a constructor would not be the copy constructor, since it takes a `Coord` pointer, not a `Coord` object. It would just be some other ordinary constructor. The copy constructor needs a `Coord` object as a parameter, but you must pass that object by reference, not by value. This

is because *it is the copy constructor that defines how pass-by-value works!!!!* If you were able to pass your argument by value to the copy constructor, you would basically be following these instructions:

```
Instructions for calling the copy constructor:
   First, since you are passing by value, you must make a copy
    of your argument, and then you can run the copy constructor
    definition code on that copy. To initialize the copy of your
    argument, you use the copy constructor -- your newly-created
    memory on the stack space is what you are trying to initialize,
    and whatever you passed by value is the argument you'll pass to
    the copy constructor to make this copy. See ``Instructions for
    calling the copy constructor'' to learn how to do this. Then,
    once you have correctly initialized the copy of your argument,
    you can take this newly-initialized copy, and run the copy
    constructor definition code on it.
```

That is, you'd end up with an endless recursive definition with no base case, and as you know, recursion without a base case is not good. Since it is the copy constructor that defines how copies should be made, the copy constructor itself cannot require that a copy be made in order to work correctly. (This is a tough issue; think about for a bit if it doesn't make sense, and perhaps try and trace a few such calls out on paper.) Therefore, you cannot pass by value to the copy constructor; you must use a reference variable as the copy constructor parameter. (A good compiler should detect and prevent your use of a by-value parameter in the copy constructor.)

1.6 Destructors

Another special function a class can have is a *destructor*. The destructor, like all constructors, has no return type. In addition, the destructor never has parameters, and it has a specific name – the name of the class, preceeded by a tilde.

```
// From now on, we'll only list functions relevant to our
//  discussion, and Initialize and Print aren't relevant to
//  our discussion, so away they go.
class Coord {
   public:
      Coord();
      Coord(double x, double y);
      Coord(const Coord& origVal);
      ~Coord();
   private:
     double xCoord, yCoord;
};
```

The destructor declaration is seen in the code above. Since you can never have parameters in a destructor, that means it's not possible to overload a destructor, and thus a class can have only one destructor, never more than that.

Destructors, as the name indicates, are responsible for helping to destroy an object. Unlike constructors, destructors are not called directly. Rather, there is an implicit call to an object's destructor when the object goes out of scope. In addition, the destructor is called implicitly when the delete keyword is called on a pointer. In that case, before destroying the object that is being pointed to, the computer calls the object's destructor. So in other words, whenever an object – local or dynamic – is about to be destroyed (i.e. have its memory given back to the system), its destructor is called first.

The job of the destructor is to clean up any resources that the object requested during its lifetime. In CS225, the only resource an object might request will be dynamic memory of its own, and in the following pages, that will be the kind of resource we will use in our examples. In general, however, an object might also request other system resources (for example, a lock on a certain piece of data) whose discussion is beyond the scope of this course. An object could require many different system resources over its lifetime, and when the object gets destroyed, any of those resources that it has not yet given back, must be given back just before the object is destroyed. It is the destructor's job to do this. A correctly-written destructor knows what resources the object *might* still have, checks to see if the object still has them, and gives them back if the object does still have them.

If you do not provide a destructor, an empty destructor (i.e. one that does nothing):

```
Coord::~Coord()
{
    // nothing
}
```

is provided by the compiler, simply so that there *is* a destructor to call. After all, if there are all these implicit destructor calls happening everywhere, you want to make sure to have some kind of destructor in existence for them to call, even if it the actual destructor that is called does nothing. So, just as with the copy constructor, you *always* have a destructor, whether you explicitly write one or not.

When do you want to write one? Well, if you have a class like the one we looked at a bit earlier:

```
class Coord {
   public:
         // whatever you want here
   private:
     double xCoord, yCoord;
     String s;
};
```

then the default destructor will be fine. The reason is that there is more that goes on during the destruction process. When the system tries to destroy an object, such a `Coord` object based on the class above, there are a number of things that happen. First, as we have discussed, the destructor for this object is called. In our case, nothing is done here, since we are using the default destructor, and that does nothing. Next, as we have discussed, the `Coord` object itself is destroyed. But what does that mean? Well, ultimately, it means the memory that `Coord` object was using – whether local or dynamic – is given back to the system. But first, the destructor of each of the member data is called! When an object gets destroyed, part of that process is

for the destructors of each of the variables to get called, to wipe out any resources the variables themselves used behind the scenes. Then, finally, the system is alerted that the memory being used by this `Coord` object is no longer needed. So our sequence is:

1. The `Coord` destructor is called. The `Coord` object itself has not made any specific request for outside resources, so the destructor for `Coord` is empty and nothing needs to be done here.

2. Then, all that is left is whatever directly makes up the `Coord` object – namely, the actual member variables of `Coord`. We proceed to call the destructor for each one of these.

   (a) The "`double` destructor" is called on the "`double` object" `xCoord`. (The system does nothing, in that case, since `double` is a primitive type and so there is nothing to do, i.e. an "empty destructor".)

   (b) The "`double` destructor" is called on the "`double` object" `yCoord`. (The system does nothing, in that case, since `double` is a primitive type and so there is nothing to do, i.e. an "empty destructor".)

   (c) The `String` class's destructor is called to clean up any resources acquired by the variable `s`. Note that this is *different* than resources acquired by the `Coord` object itself. All we know is that we have `s`, a `String` object. The details of that object are hidden from us – that is the whole idea of encapsulation. All we know is what we can do with a `String`, but not how it works. And thus we certainly don't know what resources the `String` object acquires!! Therefore, just as we trusted in the `String` copy constructor to correctly initialize a copy of the `String` object, similarly, we trust in the `String` destructor to correctly clean up after the `String` object. The `Coord` only worries about what it explicitly asks for. Any helper resources its own variables need, it leaves to its variables to properly handle. The fact that we don't need to worry about how the `String` is implemented in order to use it in the `Coord` class is a good thing; it means we can focus on using the `String`'s abilities in our class, rather than focusing on making the `String` object work or cleaning up after the `String` when we are done with it.

3. Finally, there is nothing left to do to this segment of memory that holds a `Coord` – in fact, with certain things cleaned up it doesn't even really correctly hold a `Coord` anymore. Thus, the system is notified that this memory which used to hold the `Coord` object's member variables can now be reclaimed by the system and reused in a different way.

So, that is how things work when we have a `String` object as a variable. What about when we have a `String` pointer as a variable?

```
class Coord {
   public:
      Coord();
      ~Coord();
   private:
     double xCoord, yCoord;
     String* s;
};
```

In that case, the constructor is likely going to allocate the needed dynamic `String` for `s` to point to:

```
Coord::Coord()
{
   xCoord = yCoord = 0.0;
   s = new String();
}
```

and in that case, that dynamic `String` object needs to be deallocated somehow. If we continue to have an empty destructor, well, go back and re-read the 3-step destruction sequence we discussed just above. If the destructor remains empty, where do we get a chance to deallocate this `String` object? It's not in step 1, since the destructor is empty and doesn't do anything. It's not in step 2, since now you would be calling the "pointer destructor" on `s`, and a pointer is a primitive type just like `double` and nothing is done. (That is, the system does not automatically dereference pointers or deallocate what they point to, since the system doesn't even know if they point to anything of significance, or if they point to something that for some reason needs to stay, or not. The system only worries about the pointer itself; what it points to is yours to worry about.) And by step 3, everything is assumed to be cleaned-up and the system moves on as if it is. Thus, we never deallocated the `String` and thus we have a memory leak!

The solution is to take care of this in step 1. We write an actual destructor for the `Coord` class, with actual code inside it:

```
Coord::~Coord()
{
   delete s;
}
```

and now, in step 1, the destructor is called, and calls `delete` on `s`, and thus deallocated the dynamic `String` object. As we described earlier, anything the `Coord` specifically requested on its own (and it requested this dynamic `String` in the `Coord` constructor) gets deleted by the `Coord` destructor in step 1. Then, in step 2, the variables of `Coord`, via their destructors, release resources they might have requested which the `Coord` object knew nothing about. Then, in step 3, since the `Coord` object and all its variables have released their resources, we assume everything is cleaned up and erase the `Coord` object itself from existence by returning its memory to the system.

So the rule is, if your object never obtains special resources *of its own* (we're not talking about resources its variables might request unknown to it) then your object doesn't have anything to release, so a destructor is not needed. But if that object acquires resources, then the class needs

a destructor so that each object can have instructions on how to give back the resources it has acquired.

1.7 The copy constructor, revisited

The existence or lack of a destructor is part of why a copy constructor is needed. Just as a pointer has no meaningful destructor, likewise it has no meaningful copy constructor. That is, since a pointer is a primitive type (regardless of its type; every pointer to any type is still a pointer), it is copied via the bitwise copy technique, just like `double` and `int` variables. Therefore, if you have the class:

```
class Coord {
   public:
      Coord();
      ~Coord();
   private:
     double xCoord, yCoord;
     String* s;
};
```

then the default copy constructor not only makes bitwise copies of the `double` variables, but makes a bitwise copy of the pointer variables as well. It does *not* dereference the pointer to copy what it points to; it just copies the pointer's bit pattern into the new object – and thus the new object's pointer holds the same bit pattern as the old object's pointer, and thus the new object's pointer holds the same address as the old object's pointer, and thus the new object's pointer points to the same piece of dynamic memory as the old object's pointer. That is, the default copy constructor would do the same thing as the following code:

```
Coord::Coord(const Coord& origVal)
{
   xCoord = origVal.xCoord;
   yCoord = origVal.yCoord;
   s = origVal.s;  // only the address is copied
}
```

This is known as a *soft copy*. We haven't really copied the entire object, we've just copied the top level, but any dynamic memory being used by the original object – in this case, the dynamic `String` object being pointed to by `origVal.s` – is now also being used by the new object, rather than the new object having a copy of that dynamic memory for its own use.

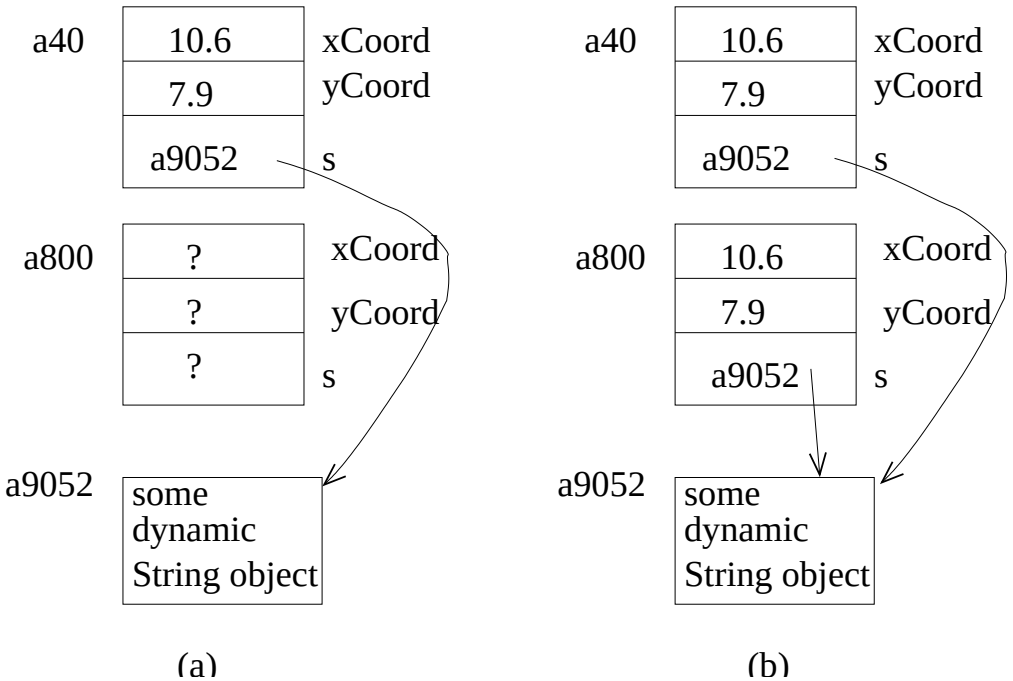

Figure 1.1: **(a)** Before running copy constructor; **(b)** After a "soft copy" copy constructor has run.

This is extremely problematic. First, any change to that dynamic memory that is made through one object results in the other object being changed as well, since they both point to the same piece of dynamic memory. We would expect the two objects to be independent, but instead changing one changes the other. Second, if we *don't* have a destructor, then neither object will delete the dynamic memory when it (the object) goes out of scope. Thus, the memory never gets freed and we have a *memory leak* as we discussed before. On the other hand, if we *do* provide a destructor, then the first object to be destroyed will delete the dynamic memory when its (the object's) destructor gets called, and now suddenly the second object has a pointer that no longer points to allocated dynamic memory, because that dynamic memory was freed by the first destructor call. This is *very* bad. One situation in which this can happen in a sneaky manner is when you pass an object by value to a function. The copy constructor is invoked, the object copy has a pointer pointing to the same piece of dynamic memory, and then when the function exits, the copy is destroyed, and with it, the dynamic memory is destroyed due to the destructor call. Now the original object's pointer no longer points to a legal allocated block of dynamic memory!

So, *that* is when you need to write your own copy constructor. Since pointers are primitive types, they are copied in a bitwise manner, when generally, what we really want instead is to create a new copy of the object the pointer points to. Since you have destructors to prevent memory leaks, you then need to make sure the copy constructor copies that dynamic memory correctly because if two objects have shared dynamic memory, the destructor for one can wipe out the shared memory and then the other object is left with nothing to point to *and the user won't know something went wrong until a segmentation fault occurs later in the program.* (When we correctly make an entirely independent copy, then that is known as a *hard copy*, in contrast

to the earlier soft copy.)

We would write this copy constructor as follows:

```
Coord::Coord(const Coord& origVal)
{
   xCoord = origVal.xCoord;
   yCoord = origVal.yCoord;
   s = new String(*(origVal.s));
}
```

First we dereference the pointer of the parameter object. That gives us the dynamic memory of the parameter object. We pass that dynamic memory into the `String` copy constructor, and use it to initialize a new dynamic `String`. And then the pointer in the new object is set to point to the new dynamic `String`.

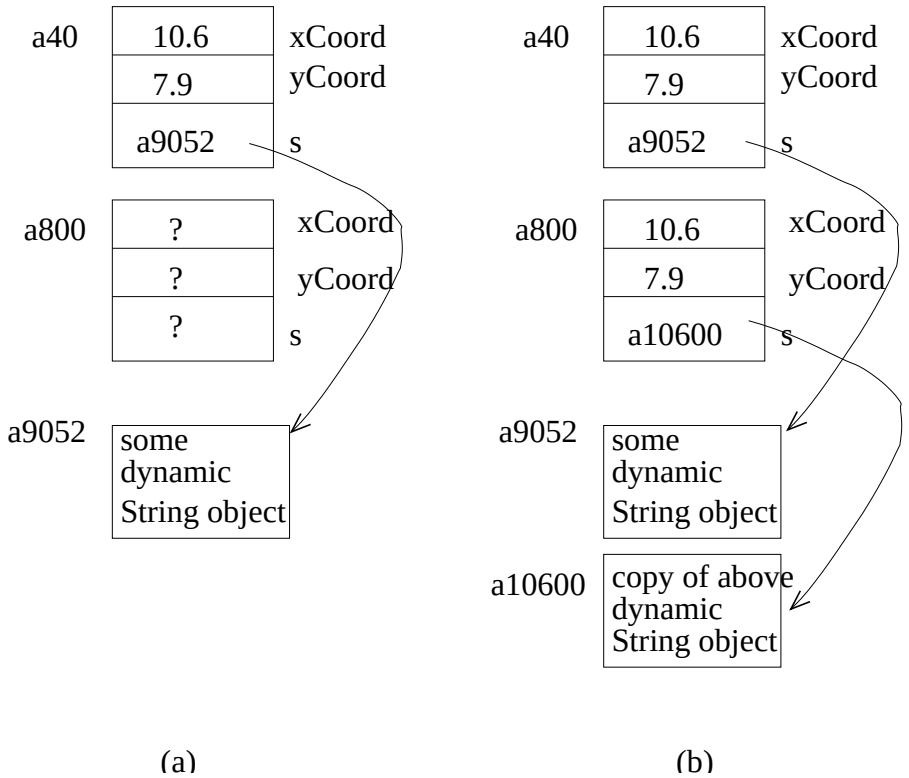

Figure 1.2: **(a)** Before running copy constructor; **(b)** After a "hard copy" copy constructor has run.

So, it is generally necessary to write a copy constructor for all your classes which have pointers as member data (i.e. classes which have member data that point to dynamic memory). Otherwise, the compiler will supply a simple one, and in the case of classes which have pointers to dynamic memory, the copy constructor for such a class is *too* simple and you will have problems.

1.8 Overloading Operators

One feature of C++ that is not found in Java is the ability to overload the operators of the language. In Java, as you might recall, you could use the `+` operator not only to add two integers, or two floating-point numbers, but also to concatenate two `String` objects. All of those abilities were built into the language. However, if you wrote a class `Complex`, to handle complex numbers, you could not use the `+` operator to add complex numbers; you would have to write an `Add` method or some such. The `+` operator knew nothing about the classes *you* wrote, and so it didn't work for classes you wrote, only classes provided by the language.

In C++, the story is different. Certainly, you can still write functions like `Add` if you want, but using the operator symbols can often be very convenient, and thus if you want to overload the meaning of an operator – for example, to define what `+` means when the two operands are of type `Complex` – you can do that. Overloading an operator is not really much different than overloading a regular function. The only difference is in how you actually name the function. Operators are just symbols that are used near variables and values, and having symbols serve as a function name would look both silly and confusing, and further, would be difficult to compile correctly:

```
+(Complex a, Complex b) // this is NOT real syntax, and is INCORRECT
```

This problem is solved by using the word `operator` to indicate an operator function. Overloading the addition operator, as in the fake, incorrect example above, would really result in the function name and parameter list appearing as follows:

```
operator+(Complex a, Complex b)
```

Other than that difference, overloading operators is really no different from overloading any other function. You can overload almost any operator in the language, which means you can have functions with names like:

```
operator[]
operator<
operator=
operator==
operator%
operator->
```

and so on.

You even have the option of using the actual function form of the operator when you call the operator function. If you did indeed overload `operator+` for a class `Complex`, and if you then had two variables `c1` and `c2` of type `Complex`, then you could use this expression in your code:

```
c1 + c2
```

but you could also use this expression in your code:

```
operator+(c1, c2)
```

And in reality, when you use the symbolic syntax (the upper one above), the compiler will just expand it to the function syntax (the lower one above). The only difference between an operator

and another function is that you *do* have the "symbol" form available to use as shorthand, instead of having to actually call a function explicitly. But when you use the symbol shorthand, the function still gets called. The shorthand is just there for the programmer's use and convenience.

Warning! This does not mean you should go crazy using all kinds of operators as shorthand. Use operator overloading *only* in intuitive ways, or no one else will ever understand your code! Using `+` for string concatenation makes sense. Using `*` or `==` for database access is not an overload whose meaning is going to be immediately obvious to anyone, and thus code littered with such use will be very confusing, and thus such operator overloading should not be done.

One final note about the syntax: another way to use these operators is for the object on the left to serve as the calling object, and the object on the right to serve as the parameter, when the function is actually called:

```
L1 = L2;      // L1 is the calling object,
              //   and L2 is the lone parameter
```

To clarify what is going on here, you should know that the above function call could just as well have been written as follows:

```
L1.operator=(L2);
```

In other words, the assignment operator function is being called off of the object `L1`, and the parameter passed to the "equals" function is the object `L2`. Just as we could call `assign()` off of some object, and pass to the `assign()` function some other object of that type, here we call the assignment operator function (which is named `operator=`) off of the object `L1` (whatever type it happens to be), and pass it a parameter `L2` of the same type. Most operators can be overloaded in the `operator(a, b)` manner or the `a.operator(b)` manner (we won't get into the pros and cons of each right now). Some, like the assignment operator, *must* be member functions and thus must be overloaded in the `a.operator(b)` form, which is also generally the more common form.

1.9 The `this` pointer

The keyword `this` is simply a "implicitly declared self-referential pointer". Meaning, when a member function is called, there is an additional local variable you don't have to declare yourself, and that variable will hold the address of the object the member function was called on. From within the function definition code, you can refer to the pointer as if you had declared it yourself. So, it is "implicitly declared", meaning it is declared and assigned automatically without any effort on your part, and it is "self-referential", meaning the pointer holds the address of the object the member function is called on. (This is more or less the same as the "`this` reference" in Java.)

```
class Coord
{
public:
   Coord LeftShift5() {xCoord = xCoord - 5; return (*this); }
private:
   int xCoord, yCoord;
}
```

In the example above, in the function `LeftShift5()`, a calculation was done on an object, and then for the return type, it was ideal to return that object, rather than creating a new one with the same information and returning that. But, how does one refer to the actual object? With the `this` pointer! In the above example, the pointer is dereferenced, giving the actual object itself. This object is the object that `LeftShift5()` was called off of, and so we would expect it to be an object of type `Coord`. This matches the function's output type, which is also `Coord`.

In fact, when you just use member data variable names in a member function, it is understood that they are the data of "this" object. So, we could have rewritten the `Initialize` function (first seen in lecture #1) as follows:

```
void Coord::Initialize(double xInit, double yInit)
{
   this->xCoord = xInit;
   this->yCoord = yInit;
}
```

or else as follows:

```
void Coord::Initialize(double xInit, double yInit)
{
   (*this).xCoord = xInit;
   (*this).yCoord = yInit;
}
```

When we aren't accessing the `xCoord` variable from some object using the dot (`.`) or arrow (`->`) member access syntax, it is assumed that we mean to access the `xCoord` of "this" object and so the effect is the same as if we had explicitly put `this` in as we did above. Or in other words, if from within a `Coord` member function, we use `xCoord` by itself, it is assumed to be the `xCoord` of "`this`"; the only way to make it clear we want the `xCoord` of `someOtherObject` is to explicitly use the syntax `someOtherObject.xCoord`. And because the system will generally assume we mean `this`, we generally don't need to explicitly use it – that is why we've never used it in any code up until this section. (Again, this is more or less the same as the "`this` reference" in Java.)

1.10 The assignment operator

Given how often assignment comes up, it is also generally helpful to have an `operator=` for a class as well – i.e. it is helpful to overload the `operator=` function for your own class. Just as with the destructor and copy constructor, if you don't provide an assignment operator, one will be provided to you by default, and just like the copy constructor, this default assignment operator will perform a memberwise copy. If you do not want a memberwise copy, you need to write your own `operator=` function explicitly – and you would decline to use the compiler-supplied default `operator=` for the exact same reasons that you would decline to use the compiler-supplied default copy constructor.

Below is the example of an assignment operator that we *didn't* need to write, i.e. the code does exactly what the default code would have done. This assignment operator is for the version of our `Coord` class where we have a `String` object.

```
class Coord {
   public:
      Coord();
      const Coord& operator=(const Coord& origVal); // declaration
   private:
     double xCoord, yCoord;
     String s;
};



// and later, in the .C file...


const Coord& Coord::operator=(const Coord& origVal)
{
   if (this!=&origVal)
   {
      xCoord = origVal.xCoord;
      yCoord = origVal.yCoord;
      s = origVal.s;
   }
   return *this;
}
```

The first line is there to make sure we don't bother to do any copying if we had invoked this function via a call such as `x = x;`, i.e. assignment of an object to itself gets ignored because the object is *already* equal to itself. This simply saves time when we have large objects that can be memberwise-copied; when we have dynamic memory involved, allowing self-assignment to proceed can actually result in errors (as we will see shortly). So, we compare the address of `origVal` – namely, `&origVal` – to the address of the object we invoked the function on. That address is stored in `this`. If we are not attempting a self-assignment, then the two addresses will be different, and we will proceed with the copying code. If the two addresses are the same, then we will skip over all the assignment code. Finally, regardless of whether we ran the assignment code or not, at the end of the function, we return the object we are running the `operator=` on, by returning the dereferenced `this` pointer. The reason `operator=` returns the object it was invoked on is so we can chain together assignment statements, as follows:

```
a = b = c = d;
```

The above gets expanded by the compiler into:

```
a.operator=(b.operator=(c.operator=(d)));
```

Each call to the assignment operator function returns the value and that return value becomes the parameter in the next call.

On the other hand, if we had dynamic memory in the class, then our class and assignment operator would appear as follows:

```
   class Coord {
      public:
         Coord();
         Coord(const Coord& origVal);
         ~Coord();
         const Coord& operator=(const Coord& origVal); // declaration
      private:
        double xCoord, yCoord;
        String* s;
   };

// and later, in the .C file...

    // Coord(), Coord(const Coord& origVal), and ~Coord()
    // definitions provided earlier in the notes

   const Coord& Coord::operator=(const Coord& origVal)
   {
      if (this!=&origVal)
      {
         delete s;

         xCoord = origVal.xCoord;
         yCoord = origVal.yCoord;
         s = new String(*(origVal.s));
      }
      return *this;
   }
```

Note that we need to delete the old dynamic memory in the assignment operator. For this reason, for the simple classes in this course the assignment operator will often simply be the destructor code followed by the copy constructor code, both wrapped in the self-assignment check and with the return line following that. Also note that, because we delete the old dynamic memory, if we didn't check for self-assignment we would (when doing self-assignment) proceed to delete the dynamic memory we wanted to copy from, since we are assigning a value to itself and so the dynamic memory in both classes is the same. So, the code will not work for self-assignment, which is why we need to check for self-assignment before entering the delete-and-copy portion of the code. (You can actually condense the code above ever so slightly by simply overwriting the pre-existing dynamic memory rather than deleting it, but I'll leave that as an exercise for you.)

1.11 The Law of the Big Three

Finally, we come to the Law of the Big Three, which states:

```
      If you need to explicitly write any one of the destructor,
```

```
copy constructor, or assignment operator rather than relying
on the compiler-supplied default, then you need to explicitly
write all three.
```

Usually, the destructor is your triggering clue. If you allocate dynamic memory in the constructors, then you need to delete it in an explicitly-written destructor – and that means that you need to explicitly write the copy constructor and assignment operator to correctly handle the copying of that dynamic memory as well. In other words, the three functions come in a set – either you need to write all three yourself, or you can rely on the compiler-supplied defaults for all three.

There are rare circumstances where this "law" might not hold, but it will generally be true.