Lecture 34

Clarifications and Corrections

I was asked about the return type on the operator= function, and why we needed one at all. The idea is that we have two existing objects, and we are copying the contents of one into the other, so what is the const ref referring to?

There is at least one (meaning, I have found one) reason for this. If we want to string together several assignments, C++ typically lets us do this. For example:

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

If `a`, `b`, and `c` are all `int`s, then the intent of the statement is straightforward. The "=" operator evaluates right-to-left, so we save the value of `c` into `b`, then save the value of `b` into `a`.

What if `a`, `b`, and `c` are objects. Then the "=" operator is overloaded to call the operator= function. We will do the `b = c` operation first. Then, we would want to do the `a = b` operation. In order for C++ to recognize that we need to call `a`'s operator= function, we need an object of the same type on both sides of the "=" sign.

We also talked about the initializer list for the copy constructor. I said that we could assign the values instead of using the initializer list, and that this would be the same thing.

In fact, if our code is written correctly, the result will be the same, but we will get there a little differently.

The initializer technique will trigger calls to the member objects' respective copy constructors. However, if we assign the values in the body of the copy constructor, this will trigger calls to the operator= functions of the member objects. The end result is the same–two objects that are copies of each other, but the method is slightly different.

Using streams

For your assignment this week, you will need to use ofstream and/or ifstream objects in C++. There is a small detail that you need to know. The following code will not compile:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

void doFileStuff (string filename) {
  ofstream myfile(filename);
  myfile << "Writing this to a file.\n";
  myfile.close();
```

```
}

int main() {
    doFileStuff("filename.txt");
}
```

The reason is that the ofstream and ifstream constructors expect a primitive string type. And we have used the C++ string object. We can get the primitive string with `filename.c_str()` as the constructor parameter.

The big 3 (cont.)

Naive approach to pointers

The code on the code page for this lecture under the directory `ptr_shallow_impl` gives a naive attempt at rewriting the class structure from last lecture using pointers to objects instead of objects.

For example, in our `BigClass` definition, we have changed the declarations of the objects to:

```
ThisClass *a;
ThatClass *b;
```

We have made a similar change to the `SmallClass` objects in `ThisClass`.

For the sake of this example, we assume that we have a basic understanding of pointers, and so we recognize that we need to create our objects that we've declared in the constructor. For example, we add:

```
a = new ThisClass;
b = new ThatClass;
```

to the `BigClass` constructor.

We also remember that any time we create an object on the heap using the `new` command, we need to be responsible for deleting the object when we are done. For this reason, we add the appropriate `delete` commands to the `BigClass` and `ThisClass` destructors. We (naively) assume that the copy constructor and operator= functions that C++ writes for us will be fine.

If we run the `class_tester.cpp` code, we see that our assumption was incorrect. First of all, we notice that we are getting a run-time error. We know that the first object to get destroyed is `b3`, because C++ will go in reverse order. The error occurs in the SmallClass destructor for our first call (s1) from object `b2`. Why?

We created our object `b`. Then we created our object `b2` by calling the copy constructor. But if we look at the output, we see that we only called the `BigClass` copy constructor, we didn't propagate down through the structure like we did with our previous object-based example.

Here's the problem: the copy constructor C++ gives us just copies each field. So we constructed a new object, and the copy constructor copied the value of `a` (a pointer to a `ThisClass` object) from the original object into the value of `a` (a pointer to a `ThisClass` object) from the new object.

This is called a *shallow copy*, because we haven't really copied the object at all. We have copied the pointer, so now we have two objects, but their `a` fields both point to the same `ThisClass` object!

Let's look at our operator= function call. Again, we see that we don't cascade down through the structure. Our call to the operator= function begins and ends with the call to the `BigClass` operator= function. We never call the corresponding operator= functions for the other classes.

If we look at the implementation, we see why. The default is to copy the value using the "=" operator. When the objects being assigned are pointers, then this just copies the value of the pointer. We get a similar problem. The `a` member of `b3` points to the same `ThisClass` object as `b` and `b2`.

In fact, this is worse than our problem with the copy constructor, because when we created `b3`, we created a complete object, with all of its member objects. When the operator= function reassigned the pointer to `a`, the `ThisClass` object that `a` originally pointed to is now still allocated in memory, but it does not have any references to it. This is a memory leak!

Both `b2` and `b3` are shallow copies of `b`. Our code crashed when we tried to delete `b2`, but would have also crashed when we tried to delete `b`, because `b3` had already deleted the `SmallClass` objects that all three were pointing to.

A better approach to pointers

We had a good start last time. We needed to initialize our objects in our constructor, and we needed to destroy them in our destructor.

The problem was that we were making a shallow copy when we used the copy constructor or the operator= function. Let's look at the copy constructor first. The copy constructor is supposed to be a constructor. So it should be making new objects for all of the components for the big class. That is, we need to use the `new` command to create a new heap allocated object. This new command should in turn call the copy constructor of the smaller objects, so that all new objects are created. We have to do the same thing in `ThisClass`, since we have pointers to heap allocated objects there, too. For our copy, we need to create new `SmallClass` objects on the heap for `s1` and `s2`.

Next, we look at the operator= function. Here, we assume we have two created objects (that are correctly created!). We want to copy the contents of one into the other. Before, we just copied the pointer of each object. This was wrong, because it created a shallow copy. It also didn't call the corresponding operator= functions, since the objects being assigned were pointers. We need to dereference the pointers, and use the "=" operator. This way, we are assigning an object to another object of the same type. This will cause C++ to call the operator= function for the smaller objects, which will call our overridden function in the class.

When we run the code, we get the same results as when we created our objects on the heap. We see the constructors creating all new objects, and the operator= function calling all the way down to the `SmallClass` objects.

More importantly, we see that the destructors behave as they should, and we don't have any shallow copies that cause problems when we destroy the objects.

I'll leave it to you to examine the code and convince yourself that we're really doing what we want.

Some C++ class odds and ends

default constructor

If we write a class with no constructor (not including the copy constructor, which C++ writes for us), then C++ will write a no parameter constructor for us. This constructor will create a new object of the class, and initialize all of it's object members using their no-parameter constructors. It is not guaranteed to initialize primitive types on the run time stack, or pointers to objects intended to be allocated on the heap (in fact, we can be sure that these will not be initialized).

disabling copy constructor and operator=

We may decide, for some reason, that we want to disable the copy constructor, the operator= function, or both. This can be achieved by moving their declarations to the `private:` section of the class definition.

Friends

Java has several modifiers for controlling who can see what members of a class–private, *no modifier*, protected, and public.

C++ has only public and private, but it has something called a *friend declaration*.

```
    class ListNode {
        private:
            int element;
            ListNode *next;

            ListNode(int theElement, ListNode *n)
                : element(theElement), next(n) {}
        friend class IntQueue;
};
```

In the above case, we have declared that the `IntQueue` class is a *friend*, and thus can see **all** of `ListNode`'s private elements.

If we use:

```
    friend void IntQueue::enqueue(int x);
```

Then we are saying that only the specified member function `enqueue` of `IntQueue` has access to all of `ListNode`'s private members.

There is **no way** to specify only certain private members can be accessed by a friend.