Lecture 33

More C++ pointers

Refer to the file `ptr_exs2.cpp` on the code page for this lecture.

In function `objectsPointers2`, we create a pointer for a vector, but do not call the `new` command. Instead, we create a new vector as an object on the stack. Then we assign the memory address of the object on the stack to the pointer we created.

Notice that the code does what we expect. We can use our pointer to reference the object created on the stack. The values we've added to the vector using the pointer show up when we print the values of the vector we created. They are referring to the same object.

Also, notice that we do not call the `delete` command in this method. This is not a mistake. Why don't we call `delete`, and how is the object deleted?

C++ pointer odds and ends

Remember that the dereference operator has lower precedence than some common operators. This can be an issue if we're not careful. Examine the `precedence` function in our `ptr_exs2.cpp` program.

Notice that the suffix `++` operator happens first. What happens when we increment the pointer, it increases `ptr` by one memory unit. That is, the pointer is a pointer to an `int`. To increment the pointer is to point to the next `int`. If an `int` is 4 bytes in size, then the next `int` is 4 bytes away.

Finally, examine the `fun` function in our sample program. The * operator dereferences an address. That is, it goes to that address and gives us the value stored there. The & operator is the address-of operator. It gives us the address of the value we are looking at.

These two are opposites of each other, and we can combine them in interesting ways. We can dereference, then "rereference." We can also get the address on the stack where we stored the pointer.

The Big Three

Every time we create a class, C++ automatically writes a destructor, copy constructor, and operator= function for us. They are hidden from us (that is, we don't see them in our class), and they are written assuming we are allocating objects on the stack. If we are, then they usually work fine without us changing them. However, if we are using pointers, and allocating objects on the heap, then we sometimes have to alter them to make sure they do what they are supposed to.

The job of the **destructor** is to clean up any memory that was allocated by our class. Since the default for C++ is to allocate objects on the stack, and since these objects are automatically reclaimed when they go out of scope (e.g., the function exits), the default destructor typically does not have anything to do.

The **copy constructor** is called whenever a new object is created and initialized to a copy of the same type of object. For example, assume that `original` is an object of `MyClass` that already exists:

```
MyClass copy = original;
```

or

```
MyClass copy(original);
```

Note that the two statements above are equivalent. That is, they are different syntax for the same thing – a call to a single parameter constructor, whose parameter is of the type `MyClass`.

The following code, however, is not handled by the copy constructor:

```
MyClass original;
MyClass copy;
copy = original;
```

The last line of this code is intended to copy the contents of one existing object into another existing object. It is handled instead by the **operator=** function.

Objects on the stack

The sample code on the code page for this lecture that is listed as being in the `./object_default` directory, describes a series of related classes that buld a complex object. A `BigClass` object consists of a `ThisClass` object and a `ThatClass` object. A `ThisClass` object, in turn, consists of two separate objects of type `SmallClass`.

Our code does not have a destructor, copy constructor, or operator= function, but C++ has written each of these for us. Unfortunately, we can't see them. We can, however, override them. So, to see what's going on, we explicitly write each of these three functions, but we implement them so that they do exactly what the default C++ functions do. That is, the code behaves exactly the same way, except that we can put some print statements in the code so we can see what's going on. We have done this, and the result is the code listed on the code page for this lecture under the directory `./object_impl`.

The default constructor (not one of the big three) sets the value of the `int` member `id`. The objects are created automatically, and do not need to be initialized. We initialize the integer just to show one way to initialize values in a constructor.

The copy constructor is a constructor that takes another object of the same type to make a copy. The signature requires that we pass the object as const reference.

There is something new here called an initializer list. We could have said:

```
a = rhs.a;
b = rhs.b;
id = rhs.id;
```

Instead, we say initialize the value of `a` to the value of `rhs.a`. So, what's the difference. Not much, as it turns out. The way listed above, we create the new member object and initialize it to its defaults, then we copy the values into the new object by calling the operator= function. With the initializer list, we use the values we give it to initialize the memory the first time using a call to the copy constructor of the member object. So, basically, it saves us a copy.

The operator= method looks similar, but is a little different of course. First, we need to test for aliases. We do this because the operator= function does this by default. Remember, we are simply implementing the identical functionality that C++ gives us. C++ does this because it needs to be generic and work for all objects.

If you consider the objects being copied are file objects, then we may cause problems if we try to copy a file to itself. Typically, when copying files, the operating system will check to see if the destination file exists. If so, then the operating system will truncate the destination file before doing the copy. If we copy a file to itself, then we may end up deleting the file.

Once we've made sure we don't have an alias, then we copy the objects and values into the current object. If we are copying an object, then the "=" sign will be a call to that object's operator= function.

Finally, we have the destructor. Since for this example, all of our objects are stack allocated, there is nothing to do.

We repeat this code, as appropriate for each of our classes. If we run the `class_tester.cpp` file, we see the behavior of the big three functions.

The first thing that we do is create a new object. This results in a call to the `BigClass` constructor, which calls the `ThisClass` constructor, which calls the `SmallClass` constructor twice, etc.

The next thing we do is create another new object using the first object. This calls our copy constructor. Again, we see that the copy constructors for each object call the copy constructors for the member objects.

Then, we create a new object, and separately set the new object equal to the first. This calls the operator= function in a similar way as before.

When the `main` function ends, our objects go out of scope, and their destructors are called.

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 above class structure 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.

Our naive approach is clearly inadequate, and needs some work.