Lecture 35

Application of pointers

We've been talking about pointers in C++, and how we can use them to write classes and code that is more like the Java code we're used to. We looked at C++'s big three class functions – the destructor, the copy constructor, and the operator= function, and how their default behavior usually doesn't handle pointers well.

Next, we're going to look at some code that applies what we've learned, and we're going to talk about some of the problems we need to be aware of.

We start with the linked list code from last week's lab. We know that this code has some problems, but it is a good place to start.

The `Node` class is defined in the `Node.h` header file, and we have separated out the implementation in the `Node.cpp` file. This is why we need to explicitly identify the class of the member functions when we implement them. (E.g., `Node::value()`).

The implementation is very straightforward. Two constructors, `next` and `setNext` for `Node`s, and `value` and `setValue` for the integer values stored in the node.

The `LinkedList` class implementation is similarly straightforward, and comes from our Java LinkedList class from Bailey.

The test function `test2` in `linkedlisttest.cpp` creates a linked list called "l" and adds the values 0 to 9 to the list. It creates another linked list called "l2" using the copy constructor.

It changes one of the values on list "l," then prints out the state of both lists by repeated calls to the `removeFirst` method

When we run this test, we get the following output:

```
L: 9 8 7 6 5 100 3 2 1 0
L2: 9 8 7 6 5 100 3 2 1 0
```

This is not what we expected. By creating `l2` as a copy of `l`, we expected that changes to `l` would not effect `l2`. However, it is clear from the output that the change to `l` effected both lists. Why? Because the code uses the default copy constructor and operator= functions. The copy constructor (called here) just makes a copy of each member of the object. In this case, that was the pointer to the head of the list. When it copied the pointer, both `l` and `l2`'s head pointer pointed to the same first node. When the item at index 5 was changed, it was the same node for each list.

Notice also that when the code prints out the contents of list `l`, it does so using the `removeFirst` member function. When it is done printing out `l`, `l` should be empty. That is, the nodes pointed to by the shallow copy `l2` should be gone. But they aren't.

When the code prints out `l2` with calls to `removeFirst`, it seems to work. The problem is that in addition to

the shallow copy mentioned above, there is also a memory leak. `removeFirst` and `clear` are both supposed to remove nodes from the list, but neither does. They both just reset the `head` pointer.

Improved LinkedList

To improve the linked list, we need to write our own copy constructor and operator= functions that properly handle the nodes on the list.

In the operator= function, we use a call to the `clear` function to make sure the current list object doesn't leak memory. We know this wasn't written correctly in our previous version, so we'll have to get back to this later.

In order to copy the list, we can't just copy the current object like we did with our `BigClass` example. We need to copy the contents of the list, too. This means our operator= function will need to walk down the list, and copy each node object that is on the list. Since our nodes are pointers, we will need to use the `new` command to create a new node for each.

An empty list is treated as a special case. Otherwise, we use two pointers. One points to the list we are copying, and the other to the list we are copying into. We create a new node for the node we know is on the list, using the value from the `rhs` passed to the `Node` constructor. We advance both pointers until the one tracking the source list reaches the end (i.e., NULL).

The copy constructor essentially does the same work, except that it creates a new object to start with. Knowing this, our implementation just calls the operator= function, and copies the source list into the current (i.e., `*this`) object.

We've also added a destructor function, to make sure that we are cleaning up the memory we've used. In this case, we just call the `clear` function, since we were already planning on writing one anyway.

These changes correctly handle our shallow copy problem. But, we have some memory leaks to take care of. `removeFirst` just advances the `head` pointer in the linked list, and the `clear` function just sets `head` to NULL.

The `removeFirst` member function is rewritten to save the old head value. It can then advance the `head` pointer, and then `delete` the node that the old head pointer pointed to.

The `clear` method is rewritten to make repeated calls to `removeFirst` until the list is empty. This way, it does not have to repeat code.

Problems

The code in `problems.cpp` on the code page of this lecture demonstrates two problems we may face when working with pointers.

First, `problem1` allocates a vector on the heap, then immediately deletes it. Next, it allocates another vector on the heap, and pushes the value 20 onto it. Then, it pushes 10 onto vector `v`, the one that was deleted. The code runs without errors. Why?

The code allocated a vector called `v` on the heap. The `new` command returned the address of the allocation and stored it in `v`. Then the code deleted `v`, but this just de-allocates the memory. `v` still has the memory address.

Next, the code allocated a new vector `v2`. The system allocated memory for `v2` and it used the same location! Since the memory that it allocated earlier for `v` was no longer in use, it was a perfectly reasonable location to store the new vector.

Now, `v` and `v2` point to the same location. We have accidentally created an alias to the vector.

In Problem2, we see a similar problem. The code sets up a special function whose purpose is to create a vector for us. Instead of allocating it on the heap, we allocate it on the stack. Since we know we want a pointer, we create a pointer and return it.

Once the function returns, the memory we used for the vector is reclaimed by the system. In short, we don't know what will happen to it. It is almost certain that it will be used for something else. In the worst case though, it may not be used right away. It may allow us to access it's values for a while before it is used again, which can cause very strange and unpredictable behavior in our code.

Style

The code in `test2` of `linkedlisttest.cpp` above is perfectly fine, and works without errors. But there is an issue with the code style. We are creating our main lists as stack allocated objects. This is nice because when we are done with them, they will be automatically reclaimed. However, we are creating all of our member objects as pointers, and somehow this seems inconsistent.

Furthermore, as we saw in the problem2 example, if we make a mistake and refer to these objects after the function exits, it may lead to bugs that are hard to find.

As a matter of style, it seems like a nice idea that if we are going to do any memory management, we should probably do all of it.

The code in function `test3` of `linkedlisttest.cpp` uses a different approach. Instead of declaring our local variables as stack allocated objects, we declare them as pointers, and explicitly allocate them on the heap. Notice that we have to use the "`->`" operator now everywhere we were using the "." operator. Other than that, though, the code works just fine.

In fact, it is even more Java like now. All of our objects are pointers, not just the ones inside our main objects. They behave just like Java, in that if we set one equal to another, then we are copying the pointer value from one variable to another.

But, we can still use the operator= function if we want to by dereferencing the source and destination objects. It's like the best of both worlds.