CMPT 225 Lab 6: Deep Copies, Destructors, and Memory Leaks

In this lab you are going to look at some of the issues that arise from using dynamic memory. The first relates to copying an existing object that has attributes in dynamic memory. This issue exists in both C++ and Java. The second issue concerns cleaning up allocated dynamic memory when it is no longer needed. This is an issue in C++ but Java does this automatically for you.

Deep and Shallow Copies

A **shallow copy** of an object is one where only the **addresses** of data created in dynamic memory have been copied, rather than the data themselves. A **deep copy** of an object is one where new dynamic memory has been allocated to create copies of any dynamically allocated data (rather than just copies of the pointers to those data).

The consequence of creating a shallow copy is that any dynamically created data is not actually copied, so that changes to one version are reflected in changes to the other.

Part 1 - Experiment

Download this zip file and save and unzip it in your lab3 directory. This zipfile contains .h and .cpp files for LinkedList and Node classes. It also contains a stub of a driver program test.cpp, and a makefile for building a program named "test."

I've also provided you with a simple test function (listTest(), below) that:

- Creates a new linked list
- Inserts some data into it
- Uses the copy constructor to create a second list
- Makes changes to the original linked list
- Prints both lists

Call this test function in your main method and satisfy yourself that the copy constructor is making a shallow copy, and that there is, in fact, only one list. You will probably need to add a few lines to the top of your file to get listTest() to work. If you don't know what is necessary, then try to compile and see what the compiler tells you. Compiler error messages are often not easy to understand, but they often tell you exactly what is wrong (but not necessarilly how to fix it.) You should practice interpreting them.

Test Function

```
void listTest() {
        LinkedList list;
        list.add(1);
        list.add(2);
        list.add(3);
        cout << "Print first list -- expecting {1, 2, 3}." << endl;
        list.printList();
        cout << endl << endl;

        cout  << "Make a copy of the list" << endl << endl;
        LinkedList secondList(list);

        list.add(4);
        list.add(5);
        list.add(6);

        cout << endl << "Add items and print first list -- expecting {1,
        list.printList();
        cout << endl << endl;

        cout << "Print second list.  If it was deep copied should be {1,
        secondList.printList();
        cout << endl << endl;
}
```

Part 2 - Deep Copy

Change the copy constructor so that it creates a deep copy of the list. Use your test program from Part 1 to satisfy yourself that the copy constructor is making a deep copy (i.e. that changes made in one of the lists are *not* reflected in the other list).

If you have successfully completed the deep copy, please ask a TA to take a look at this output, and receive your mark for this lab.

You should also complete the rest of this lab below, but this can be done as homework if you do not have time to complete it in your lab time slot.

Destructors

Every class that uses dynamic memory (like our LinkedList) requires a **destructor** to de-allocate the dynamic memory when it is no longer needed. The destructor is not called directly but it is invoked whenever an object is deleted (using **delete**), and on various system events (such as the program terminating). If all pointers and references to a piece of dynamic memory are deleted, then the memory should be de-allocated. If it is not de-allocated, then it is called a *memory leak.* Memory leaks can slow and ultimately crash long-running programs.

In our case, the only pointer to the first Node of the LinkedList is held by the LinkedList. So when the LinkedList is deleted, we must delete the first Node. The destructor for LinkedList is where we should do this.

However, note that the only pointer to the second Node in the LinkedList is held by the first Node in the LinkedList. So when the first Node is deleted, we must delete the second Node, and so on, to the end of the list of Nodes. There are two approaches to this: one can recognize in LinkedList that all Nodes in the list need deletion, and so delete them all with a loop in the LinkedList destructor. Or, one can delete only the first Node in the LinkedList destructor, and then delete a Node's *next* Node in the Node's destructor. (But do not use **delete** on a null Node--you'd have to check for that.)

The two methods for destruction above are *iterative* (the loop in the destructor ~LinkedList()) and *recursive* (each Node deletes the next Node). These are two classic different views of a linked list. Almost any whole-list operation on a linked list can be implemented in these two ways.

Anyhow, the destructor for a class should destroy (using **delete**) any and all objects that it dynamically created (with **new**).

An exception to the above rule is when an object is explicitly creating the dynamic-memory object for some code that calls it. One can usually, but not always, detect this by looking for a dynamic-memory object that is the subject of a ***return*** *statement.*

For example, you might have client code that creates an object of class Architect. It then asks the Architect to create an object of class Blueprint. The Architect would create a Blueprint (in dynamic memory) and return it to the client. Having gotten the Blueprint, the client has no further use for the Architect, and so it deletes the Architect. In this case, the Architect should not delete the Blueprint that it created with ***new****, as that Blueprint was returned to the calling code.*

Part 3 - Destructor

Write the destructor for the linked list class. You may use either the iterative or the recursive method to delete the Nodes. You never directly call a destructor, C++ calls it for you if you use **delete**, or automatically on objects when they go *out of scope* at the end of a block of code or when the program terminates.

*For almost all variables, the curly braces ('**{**' and '**}**') most closely around the variable definition denote the* ***scope*** *of the variable. Inside these curly braces, and after the variable definition, you can use the variable. Outside of the curly braces you cannot. When program execution reaches the point where it leaves the block surrounded by the curly braces, the variables inside are said to* ***go out of scope****. For example, in:*

```
bool makeCorner() {
        Figure figureOne;
        figureOne.clear();

        if( isDrawing() ) {
                Path *currentPath = new Path();
                currentPath->forward(1);
                currentPath->turnClockwise();
                currentPath->forward(1);
        }
        figureOne.close();

        return true;
}
```

the variable currentPath goes out of scope when the "true clause" of the if-statement ends (after the second "currentPath.forward();"). Since currentPath is a ***pointer****, C++ does not call a destructor at this point. (Pointers do not have destructors.) The only reference to the Path that was allocated using* ***new*** *is gone now, but that path was not de-allocated using* ***delete****. This creates a memory leak at the end of the if-statement.*

To correct the memory leak, one should insert the statement "delete currentPath;" after the second "currentPath->forward(1);".

The variable figureOne goes out of scope when the subroutine ends. However, figureOne is a Figure (an object), not a pointer to a Figure. Since it is an object, C++ will call its destructor at the end of the subroutine when it goes out of scope.

Testing Your Destructor

We will use a program called **valgrind** to check for memory leaks. Run your exectuable called test through valgrind:

```
uname@hostname: ~$ valgrind -q --leak-check=full test
```

or

```
uname@hostname: ~$ valgrind -q --leak-check=full ./test
```

(the use of ./ depends on your environment configuration, either you need it or not)

If your destructor has been implemented correctly you won't see anything unusual in the console, if you have a memory leak you will see something like this:

```
uname@hostname: ~$ valgrind -q --leak-check=full test

Print first list: {1,2,3}
{1,2,3}
Make a copy of the list

Add items and print first list: {1,2,3,4,5,6}
{1,2,3,4,5,6}

Print second list, if it was deep copied should be: {1,2,3}
{1,2,3}

==26216== 48 (16 direct, 32 indirect) bytes in 1 blocks are definitely l
==26216==    at 0x100011D7B: operator new(unsigned long) (vg_replace_mal
==26216==    by 0x1000013CA: LinkedList::LinkedList(LinkedList const&) (
==26216==    by 0x100000FAF: listTest() (in ./test)
==26216==    by 0x1000010D9: main (in ./test)
==26216==
==26216== 96 (16 direct, 80 indirect) bytes in 1 blocks are definitely l
==26216==    at 0x100011D7B: operator new(unsigned long) (vg_replace_mal
==26216==    by 0x100001309: LinkedList::add(int) (in ./test)
==26216==    by 0x100000F2A: listTest() (in ./test)
==26216==    by 0x1000010D9: main (in ./test)
==26216==
```