CMPT 225 - Lab 3 - Deep Copy

If you do not have time to complete this lab during the lab session, you can finish it on your own either at home by remotely login into a CSIL workstation or by going to the CSIL lab.

Observation from Lab 2

In Lab 2, you were asked to write **sum_array.cpp** and **circle_array.cpp** and both of these client code were assigned more than one responsibilities:

- The responsibility of interacting with the user (prompt user, read in what user enters on command line, validate what user has entered, etc...), and
- The responsibility of managing the data in the data structure **array**.

This situation can be improved by assigning the responsibility of managing the data (elements) to a data collection class designed and implemented as an abstract data type (ADT) class

One such data collection is the **List**. The List is a very flexible data collection as it can be position-oriented or value-oriented.

- A **position-oriented** List manages it elements by focusing on the position of each of its elements in its underlying data structure (or concrete data type - CDT). For example, in such List, client code would indicate at which position an element must be inserted into the List when calling the List's **insert(...)** public method: **theMembers->insert(position, theNewMember)** where **theMembers** is a pointer to an object of the List class and **theNewMember**, an object of a class modelling a member such as the Profile class.
- A **value-oriented** List manages it elements by focusing on the value of one (or more) attribute (i.e., data member) of its elements. We shall refer to this attribute as the **search key**. This search key is used to manipulate the elements of the List. Often, a value-oriented List keeps its elements sorted (but not always) using the elements' search key. For example, in such a sorted List, the insert method would first need to establish where the new element is to be inserted in order to keep this List sorted, then perform the insertion. The client code would call the List's **insert(...)** public method as follows: **theMembers->insert(theNewMember)**.

Because a List may require a large amount of memory to hold all of its elements, its implementation would often use **heap-allocated** concrete data type (CDT).

Heap memory (also called **dynamic memory**) is an alternative to **stack memory**. Local variables are allocated automatically on the stack frame assigned to a function/method when it is called (executed), and they are deallocated automatically when the function/method exits (when the function/method's stack frame is released). Heap memory is different: one must explicitly allocate the needed memory and must explicitly release it.

In this lab, you are going to look at some of the issues that arise from using dynamic memory.

The first issue relates to copying an existing object that has data members in dynamic memory. This issue exists in both C++ and Java.

The second issue concerns the release of explicitly (dynamically) allocated memory when it is no longer needed. This is something that is done automatically for you by Java, but not in C++.

Compiler-Generated Special Member Functions

Before we proceed, let's review one important feature of C++ (taken from this source as well as this source):

The C++ compiler automatically generates a default constructor, copy constructor, copy-assignment operator, and destructor for a type (class) if it doesn't declare its own. These functions are known as the special member functions.

These compiler-generated special member functions (or methods) work just fine when our class only uses automatically allocated memory. We do not have to write our own (even though we often choose to write our own default constructor).

However, when our class uses dynamically allocated memory, these compiler-generated special member functions are not sufficient because they do not deal with the dynamically allocated memory and therefore, we need to write our own.

Shallow Copy and Deep Copy

Let's start with the first issue: copying an existing object that has data members in dynamic memory. Let's see how this can be properly dealt with when implementating a data collection List ADT class.

But first, let's define a few terms:

- **Shallow Copy**: Wiki says that "Shallow copy involves creating a new, uninitialized object, B, and copying each field (data member) value from the original, A. If the field value is a primitive data type (such as int), the value is copied such that changes to the value in B do not affect the value in A." If the field value is the memory address of an object, the memory address is copied, "... hence referring to the same object that A does. Changing the state of the inner object affects the emergent state of both A and B since the objects are shared."
- **Deep Copy**: A deep copy of an object involves creating a new, uninitialized object, B, and copying each field (data member) value from the original, A. If the field value is a primitive data type (such as int), the value is copied (as described in Shallow Copy above). If the field value is the memory address of an object/data, new memory is dynamically allocated for this field in B and the object/data, stored at this memory address in A, is copied into the new dynamically allocated memory in B.

The consequence of creating a shallow copy of an object, such as a List object, that has data members in dynamic memory, is that these dynamically allocated data will not be copied, but shared, such that changes made to one List object are reflected in the other List object.

The consequence of creating a deep copy of an object, such as a List object, that has data members in dynamic memory, is that these dynamically allocated data are copied, not shared, and changes made to one List object are not reflected in the other List object.

You will find more information regarding shallow and deep copy in the file **List.h**, which you will be introduced to in the next section of this lab.

Part 1 - Experiment

Download this zip file and unzip it in your **sfuhome/cmpt-225/Lab3** directory.

This zip file contains .h and .cpp files for a data collection List ADT class called **List** and for a **Node** class. The List class is a linked list-based implementation of a data collection List ADT class. Its underlying data structure is a singly-headed (SH - only one head) singly-linked (SL - only one link) list. This List class allows nodes (containing data) to be added to the linked list, and allows the linked list to be printed. Its copy constructor performs a shallow copy and its destructor has not been implemented yet. The zip file also contains a stub (i.e., empty) test driver program called **test.cpp** as well as a **makefile** for building a program called **test**.

Open **List.h** and **List.cpp** and read the explanations and code found in these two files.

Open **Node.h** and **Node.cpp** and read the code found in these two files.

Below you will find a simple test function called **listTest()** that:

- Creates a new List object.
- Inserts some data into it.
- Uses its copy constructor to create a second List object.
- Makes changes to the original List object.
- Prints both List objects.

Copy this function into **test.cpp** and call it from the main method of **test.cpp**.

Note: Once you have done this and before you can successfully compile your **test.cpp**, you will need to ...

1. Add additional statements to your **test.cpp**, and
2. Fix one bug in the function **listTest()** you have just copied.

Make sure you understand why it is a bug and how you can fix it.

Once you have finished implementating and debugging test.cpp, **make** it and execute it, hence confirming that the copy constructor is making a shallow copy, and that there is, in fact, only one linked list and that this linked list is shared by both List objects.

Test Function

```
void listTest(){

   List * ls1 = new List();

   /* Test Case */
   cout << "Appending 1,2,3 to the first List object." << endl;
   ls1->append(1);
   ls1->append(2);
   ls1->append(3);
   cout << endl << "Printing the first List object." << endl;
   ls1->printList();
   cout << endl << "Does it contain {1,2,3}?" << endl;

   cout << endl << "Make a copy of the List object." << endl;
   List * ls2 = new List(ls1);
   cout << endl << "Print second List object." << endl;
   ls2->printList();
   cout << endl << "Does it contain {1,2,3}?" << endl;

   cout << endl << "Append 4,5,6 to the first List object." << endl;
   ls1->append(4);
   ls1->append(5);
   ls1->append(6);
   cout << endl << "Print first List object." << endl;
   ls1->printList();
   cout << endl << "Does it contain {1,2,3,4,5,6}?" << endl;

   cout << endl << "Print second List object: (what does it contain?)." << endl;
   ls2->printList();

   cout << endl << "If it was shallow copied, it should contain: {1,2,3,4,5,6}.";
   cout << endl << "If it was deep copied, it should contain: {1,2,3}." << endl;
   cout << endl;

   return;
}
```

Part 2 - Deep Copy

Change the copy constructor such that it creates a deep copy of the List object. Use your test program from Part 1 to confirm that the copy constructor is indeed making a deep copy (i.e. that changes made in one of the List objects are **not** reflected in the other List object).

Destructors

Let's now move on to the second issue: the release of explicitly (dynamically) allocated memory when it is no longer needed.

Every class that uses dynamic memory (like the List class of this lab) requires a **destructor** to deallocate the dynamic memory when it is no longer needed. The destructor is not called directly but it is invoked implicitely when the **delete** (or **delete [ ]**) operator is called.

If dynamic memory is not deallocated, it cannot be re-used i.e., re-allocated, until the application terminates. This situation is known as a **memory leak**. On our ***target machine***, one can use a program called **valgrind** to check for memory leaks.

To do so, run your exectuable called **test** through **valgrind** as follows:

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

Since you have not yet implemented a destructor for the List class, there will be a memory leak when your **test** executes and **valgrind** will let you know by producing an output like this:

```
uname@hostname: ~$ valgrind -q --leak-check=full ./test
Appending 1,2,3 to the first List object.

Printing the first List object.
{1,2,3}
Does it contain {1,2,3}?

Make a copy of the List object.

Print second List object.
{1,2,3}
Does it contain {1,2,3}?

Append 4,5,6 to the first List object.

Print first List object.
{1,2,3,4,5,6}
Does it contain {1,2,3,4,5,6}?

Print second List object: (what does it contain?).
{1,2,3}
If it was shallow copied, it should contain: {1,2,3,4,5,6}.
If it was deep copied, it should contain: {1,2,3}.


==78007== 64 (16 direct, 48 indirect) bytes in 1 blocks are definitely lost in loss record 8 of 9
==78007==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==78007==    by 0x10936F: listTest() (in /.../test)
==78007==    by 0x109603: main (in /.../test)
==78007==
==78007== 112 (16 direct, 96 indirect) bytes in 1 blocks are definitely lost in loss record 9 of 9
==78007==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==78007==    by 0x109241: listTest() (in /.../test)
==78007==    by 0x109603: main (in /.../test)
==78007==
```

Part 3 - Destructor

Let's now fix this memory leak with a destructor. A destructor destroys all dynamically created objects (that is, all objects created using **new**).

Write a destructor for this List class. The destructor should traverse the list using **delete** to delete each Node object in turn. Note that in this lab, our data is of data type **int**, so no need to destroy it. However, if our elements were dynamically-allocated objects of some class (for example, of Profile class), i.e., if what was stored in the Node class' data member called **data** was a pointer to a dynamically-allocated object of some class, then these objects would also need to be explicitly destroyed.

Note that you never directly call a destructor, it is called for you when 1) you call **delete** (or **delete [ ]**) on a dynamically allocated object or 2) when an automatically allocated object (that has been declared on the stack memory) goes out of scope at the end of a function.

Testing Your Destructor

Use your **test** program from Part 1 along with **valgrind** again to make sure your destructor works:

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

If your destructor has been implemented correctly you should see the expected result:

```
uname@hostname: ~$ valgrind -q --leak-check=full ./test
Appending 1,2,3 to the first List object.
```

```
Printing the first List object.
{1,2,3}
Does it contain {1,2,3}?

Make a copy of the List object.

Print second List object.
{1,2,3}
Does it contain {1,2,3}?

Append 4,5,6 to the first List object.

Print first List object.
{1,2,3,4,5,6}
Does it contain {1,2,3,4,5,6}?

Print second List object: (what does it contain?).
{1,2,3}
If it was shallow copied, it should contain: {1,2,3,4,5,6}.
If it was deep copied, it should contain: {1,2,3}.
```

If your destructor has not been implemented correctly, **valgrind** will let you know by appending its output message to the result (output) of your test.

If you are curious ...

... about constructors and destructor and are wondering when they are executed, you may wish to put the following statements

```
cout << "Default constructor called!" << endl;
```

and

```
cout << "Copy constructor called!" << endl;
```

in your constructors and

```
cout << "Destructor called!" << endl;
```

in your destructor to get a sense of when these methods are called.

Warning: If you do add the above statements to your constructors and destructor to better understand when these are executed, make sure to remove these output statements before you submit any assignments.

The End

You should now be able to design and implement a data collection ADT class that uses a concrete data type (CDT) made of dynamically allocated memory and provide most of its necessary methods, i.e., **special member functions**. We say "most" because we still need to add the **overloaded assignment operator** to our List ADT class. We shall introduce the concept of overloading operators in a future lab.

Stay tuned!

Resources

Here is a good Online Tutorial describing shallow and deep copying, copy constructor as well as the overloaded assignment operator.

Enjoy!
