Lab 3 - Encapsulation and Memory Management

Goals

- In this lab, you will experience:
    - More practice with classes
    - Constructors/Destructors
    - lvalue and rvalue references
    - New/Delete
    - Management of resources using the Resource Acquisition Is Initialization (RAII) C++ idiom
    - Copy and move semantics for memory management

Introduction

- We introduce **libraries**, **include paths** and **linking libraries** in this lab.
    - Libraries are essentially "archives" of pre-compiled code that is ready for use in other code. In C++, they are essentially many different **.o** files put together to form one **.a** file which is used in a special way compared to just using **.o** files.
        - All of your **.o** files will end up being bundled in a library file with the file format suffix **.a**
    - You will also need to understand what the `-I` flag does in **g++** and the `-L` flag as well. You will also need to know what the `-l` flag does as well.
- The `ListNode` class is provided to you with a working test-ListNode.cpp file and executable -- however, you will now be linking **libContainer.a** into the final executable for **test-ListNode.out** instead of **ListNode.o**.
- For the **Vector** and **List** classes, you are *specifically* expected to implement the default, copy, and move constructors.

Task 1

- Your task is to fix the **Makefile** in the base of the directory such that it will build **./lib/libContainer.a** with **ListNode.o**, as well as **Vector.o** and **List.o** when you complete Tasks 2 and 3.
    - The Makefile rule for building **test-ListNode.cpp** is already intact, as are the test C++ files for the other two classes. In fact, the **g++** compilation commands are very similar for building the **libContainer.a** file
    - You will be using the **ar** program to build the file. For example, if we have three files called **A.o**, **B.o**, and **C.o**, and want to build a static library file called **libABC.a**, you will run the command:
        - `ar rcs libABC.a A.o B.o C.o`
- **You will not be able to complete this lab without completing this Makefile!** This is your opportunity to understand how Makefiles work!
    - Look to the other rules already included within the Makefile for guidance.

    - You may also want to lookup your own resources or tutorials on the Internet for this part.
- When you are done with Task 1 alone, you should be able to build and run **./bin/test-ListNode.out** by running:
    - `make test-ListNode`

Task 2

- Your task is to implement the appropriate member functions/methods on the class **Vector** found in **./include/Vector.hpp** in **./src/Vector.cpp** (create the file).
    - **Vector** is meant to implement a growing/shrinking array that re-allocates memory when it wants to insert an element that won't fit in its current capacity.
        - Use **new int[<some value here>] / delete**[]
    - The main challenge of this part is to make sure that you can:
        - Correctly allocate and deallocate memory
        - Know how to re-allocate memory to form a larger array
        - Keep the **size()** method/member function consistent with the actual size of the internal array.
- Add **Vector.o** to **libContainer.a** when you build the static library file in the Makefile
- When you are done with Task 1 and Task 2, you should be able to build and run **./bin/test-Vector.out** by running:
    - `make test-Vector`

Task 3

- Your task is to implement the appropriate member functions/methods on the class **List** found in **./include/List.hpp** in **./src/List.cpp** (create the file).
    - **List** is meant to implement what is called a "linked list" data structure using the **ListNode** class as seen before, and contains almost the same exact member functions as **Vector.**
        - So, **List** and **Vector** are meant to provide the same "behavior" of a self-managing "sequence" of items (sort of like a self-managing array), but do it in two different ways under the surface. Even the **test-List.cpp** and **test-Vector.cpp** files are extremely similar!
    - Each **ListNode** contains a pointer named "next" that points either to the next **ListNode** in the linked list, or is set to **nullptr** which means it is the last node in the list. This is how you will store the entire sequence of items.
- When you are done with Task 1 and Task 3, you should be able to build and run **./bin/test-List.out** by running:
    - `make test-List`

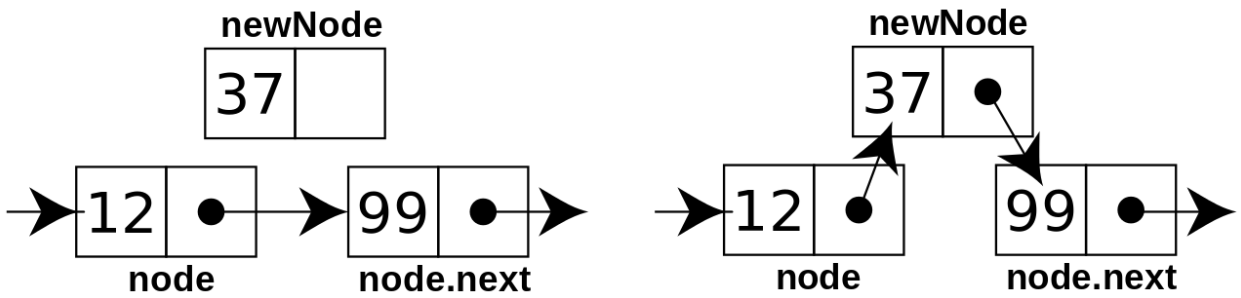

Figure 1 - An example of an insertion of a ListNode into a linked list of ListNode

Submission

- In a file called **team.txt**, write your group number on the first line. On subsequent lines, write the last, then first names of your team members separated by a single comma, each on its own line. If your name contains whitespace (e.g. the single space in the last name, "da Silva"), you may keep the whitespace.
- Your final executable should have the filename **a.out**.
- **Comment your code! If your code does not work correctly, you may be able to get *partial credit* based on your thinking that we can infer from the comments!**
- Place your source code files into a directory called **project**, using the following directory structure:
    - `project/`
        - `Makefile`
        - `docs/`
            - *`team.txt`*
        - `src/`
            - `<put your source files here>`
        - `bin/`
            - `<your executables should appear here after invoking the Makefile>`
- Before turning in your .tar file, type **"`make clean`"** in the **project** directory to remove any executable files.
- Archive and compress your **project** directory using the command:
    - `tar czf name_of_archive_file.tar.gz project`
    - The name of the archive file should be in the following format:
        - `ee205-lab<lab-number>-group<group-number>.tar.gz`
- Submit the *tar.gz* file with one group member on Laulima for the assignment.
    - For your safety, test uncompressing and unpacking your archive in a clean folder, then compile your source files. Ensure that the executable will run correctly.

Supplementary Material

Encapsulation

- In this lab, **Vector** and **List** encapsulate all the work needed to manage the internal memory of the object.
    - They both expose the operations that "simulate" or "adhere" to the idea of a "dynamically allocated sequence of objects" that can handle resizing automatically.
    - new/delete is not required to use Vector or List. Instead, the new/delete operations are "encapsulated" within the class itself -- *especially* within the constructor and destructor
- **Encapsulation** demands that you hide the internal fields of data, but it can also speak to the idea that you attempt to hide complexity from the user.
    - Here, we are encapsulating the details of memory management for both **Vector** and **List**. The user of the class does not need to learn anything about **new** or **delete** but can still use the class through the methods.
    - This is part of the power of encapsulation -- you can "hide noisy details" from the user of your object, and simplify the users' view of how the code works.
        - **This is a major part of the power of Object-Oriented Programming (OOP)!**

Limited Power & The Stack and Heap

- All current physical computers have resource limitations.
    - Power, speed, size, memory, etc.
- Programs are often measured in two ways:
    - Time
    - **Space (memory)** -- we'll focus on this one
- Most variables are allocated as a part of the program execution automatically because the size of variables are known at compile-time. This is known as "**static allocation** of memory," and this memory usually grows and shrinks in what is called "**the stack**" since its structure is closely related to the **function call stack**.
    - However, if your program **does not** know the size of memory it needs during a certain part of the program (e.g. how big a user input will be), it must ask for more from its operating environment (e.g. Windows, Linux, Mac OS X). This is called "**dynamic allocation** of memory," and this memory usually comes from what is called "**the heap**," since it usually has no true inherent structure and is perceived as just "lying around in a heap." StackOverlfow

Beautiful Balance, Deadly Asymmetry

- Imagine a world in which:
    - ...no books borrowed were never returned
    - ...no debt was ever repaid

    - ...no water used was ever put back into the ecosystem
    - **...no system resources were never returned to the operating system**
- **Resources would suddenly just disappear from view, never to return!**
- In the case of memory, taking too much and never returning it will mean:
    - Your program is “permanently borrowing” (stealing) memory from the operating system
        - If it keeps borrowing, it’ll eventually steal all the memory
            - If it steals all the memory,
                - Your other programs on your computer could run out of memory and crash
                - All the other programs on your computer could come to a deadlock because none of them crash but they can’t continue execution because they needed more memory
                - ...Who knows?
- **All healthy programs follow “the circle of life” for memory management:**
    - Acquisition
    - Usage
    - Release

Mimicry: RAII

- C++ provides a “design pattern” -- one almost universally taught and followed for C++ called **Resource Acquisition Is Initialization (RAII)**.
- Essentially:
    - Resource Acquisition Is **Initialization** (RAII)
    - Usage is Variables still in **Scope**
    - Resource Release is **Destruction**
- Furthermore, your C++ memory management lifecycle should look something like this:
    - Initialization (Constructor executes)
    - Usage (Object is used)
    - Destruction (Destructor executes)

Never Use Raw new/delete

- Because of RAII, most raw uses of “new/delete” and as well as “malloc/free” are frowned upon. Your classes should contain all it needs for memory management within the constructor/destructor if you use RAII! So: use new and delete internally within classes, and let the design pattern hide it.
    - Note: malloc/free do not call the constructor/destructors automatically like new/delete.

Copy vs. Move

- In C++11, the 2011 standard of C++, a new type of resource “semantics” or memory management behavior was added: move semantics, or the ability to directly “move” ownership of a resource from one object directly from one to another without copies.

- Thus: there exist 2 types of references in the language: lvalue vs. rvalue references.
    - Lvalues are “left-hand-side” values, or “variable places.”
    - Rvalues are “right-hand-side” values, or “raw literal values.”
    - Ex. int i = 1 ⇒ (i is an **int&** (lvalue), 1 is the **int&&** (rvalue))
- Lvalues can be “forced to become rvalues” in order to trigger the “move constructor” version of a constructor that assumes that the variable you’re passing it can be treated as a temporary. In other words, this conversion allows you to make lvalues defenseless to C++-legal “resource stealing,” allowing you to “move ownership of resources” around.
    - Essentially, it means you can copy pointers and set the original pointer to nullptr instead of copying the actual objects and then deleting the original.
        - Move becomes efficient when copy means copying objects that are like 2 kB large.
    - To do this, you use the **std::move** function on the lvalue (or rvalue) in question.
    - Read: rvalues allow for “stealing” of ownership of resources (legally), which is equivalent of moving ownership directly around.
- You have the choice of directly implementing, overriding, or deleting (yes, deleting!) copy and move constructor. This means you have control over whether an object is copyable or moveable.

Why Do I Care About Move vs. Copy?

- Move allows for “unique references” to be implemented such that there exists only “one unique” user of a resource.
    - Code: `Foo(const Foo& other);`
- Copy allows for “shared references” to be implemented such that many people can have equal access while keeping the memory management encapsulated (thanks to RAII).
    - Code: `Foo(Foo&& other);`