CS 15 Lab 1: ArrayLists

Introduction

This lab covers ArrayLists, a topic which should sound familiar from CS 11. Arrays are a great data structure because they can be indexed, allowing content to be accessed in constant time. However, a normal array must have a size declared at the start and cannot expand to accommodate extra data. One solution is to declare a really big array to start, which you know will always be able to hold the maximum amount of data. The dilemma is that you have to pick a huge size and have a huge waste of space or pick a reasonable number and risk not having enough room. In short, you can't always know in advance how much space you'll need, and it's better to let the program acquire space as necessary. ArrayLists solve that problem by expanding and shrinking as data is inserted and removed. They can accommodate as much data as needed, and they can shrink to minimize wasted space. This lab uses an ArrayList class to keep track of a shopping list. The class currently lacks the capability to expand and shrink. An expand (or shrink function) needs the following basic steps:

1. create a new array of the correct size
2. copy data into the new array
3. delete the old array
4. update the object's state

Keep these in mind as you work on the lab!

Getting Started

Note: Don't copy and paste the commands here! If you take a few seconds to type them each time for a few weeks, then you will start to learn how

they work, and this will build your intuition about how to work in our Linux environment.

- You should have a `cs15m1` directory within your home directory. If you don't, create it now with `mkdir cs15m1`, and move inside it with `cd cs15m1`.
- Create a lab1 directory inside `cs15m1` and move into it.
- Copy the starter code with the copy command:

```
cp /comp/15m1/files/lab01/* .
```

- If you're working on your local system, copy the files from the Halligan server to your local system.
- Questions: What does the . mean? Do you know how to look at the files in the course files directory and your directory and make sure you got them all?
- Remember that a good policy when programming is to write a little, compile, test, repeat.
- In `CS 15`, we'll use make and an associated `Makefile` to compile and link programs. You don't need to know how to write `Makefile` just yet, just how to use them to build a program (type `make shop`)
- Execute the program you built by running `./shop list.txt`
- Once you are done with the lab you can submit the files with provide. The precise command is at the bottom of this handout. Remember that this time you need to submit a `README` file together with all of the source code.

The Lab

In this lab, you will work on a program that tracks a shopping list at the grocery store. The program keeps track of two lists: one list of items to buy (the shopping list) and one list of items that have been bought. The program will populate the shopping list from the file given on the command line, then the user (you!) can type items to move them from the shopping list to the purchased list. The starter code works as it is now, but only for short lists. Try compiling and running the program with `./shop shortList.txt` to see

how it works. `main.cpp` contains all the functional implementation for the grocery list program, using `ArrayList` class as the underlying data structure. The `ArrayList` class contains all standard public functions, including a constructor and destructor:

```
#include <iostream>
#include <string>

class ArrayList {
    public:
        ArrayList();
        ~ArrayList();

        void insert(std::string item);
        bool remove(std::string item);
        int  find  (std::string item) const;

        void print(std::ostream &out) const;
        int size() const;
    ...
    private:
    ...
};
```

You may add private functions, but **DO NOT** change the public function contracts. `main.cpp` creates two lists, `toBuy` and `bought`, to keep track of the shopping list. When the user types an item, exactly as it appears on the list, the item moves from `toBuy` to `bought`. At any point, the user can type `Print` or `Quit`.

Part I: Expanding and Shrinking

Your friend would like to use this program to shop for their annual, end-of-summer barbeque, for which they will need to buy A LOT of food. They gave you a text file with their list, `bbq.txt`, but currently the program cannot handle such a long list. Try executing the program with `bbq.txt` to see what happens. Your first task is to allow the program to handle longer lists by completing the `expand()` and `shrink()` functions in the `ArrayList` class. Both functions should change the capacity by a factor of two; expanding the list should double the capacity, while shrinking should halve the original capacity. You may write helper functions if you want (as long as they are private). You also should decide when to call `expand()` and `shrink()`. How will you know when the array is full (or too empty)? Note: The program will automatically quit when the shopping list is empty.

Part 2: Resize

You might notice that `expand()` and `shrink()` look pretty similar. In computer science, we are always trying to be as efficient as possible with code. Rather than a function for making the array bigger and a function for making the array smaller, we could write a single function that changes the size of the array to a given parameter. Write a function, `void resize(int newSize)`, that changes the size of the array to the given parameter and change `expand()` and `shrink()` to call this new function.

Part 3: Checking for Memory Leaks/Errors

By constantly expanding and shrinking, `ArrayList`s create a lot of opportunities for memory leaks. To check that code for leaks, we can use `valgrind`. This is a tool for checking for memory leaks and memory errors that you should use on every assignment for this class (we use it to grade your assignments). The handout that you read in advance of this lab provides a good overview for how to interpret the output that the tool produces. To run `valgrind`, simply write `valgrind` before the executable. For example, `valgrind ./shop shortList.txt`.

Valgrind and OSX

Note! If you are working on a `Mac`, then `Valgrind` will **not** work as expected. You can do one of the following two things:

- Copy the files from your local system to the Halligan server and run `Valgrind` there.
- Or, use the `AddressSanitizer` compilation option. To do this:
  1. Open your `Makefile` and follow the instructions there.
  2. Run the command `whereis llvm-symbolizer`. If the command is not found, you need to install it. See this link for details `https://embeddedartistry.com/blog/2017/02/24/installing-llvm-clang-on-osx/`. Once you have things ready, running `whereis llvm-symbolizer` should produce something like this:

```
mrussell:lab01$ whereis llvm-symbolizer
llvm-symbolizer: /usr/bin/llvm-symbolizer /usr/share/
    man/man1/llvm-symbolizer.1.gz
```

3. Now, open the file the runs on your login; by default, on a `Mac` this will be `~/.zshrc`. At the bottom, add the following line:

   ```
   path+=('/usr/bin/llvm-symbolizer')
   ```

   Where you replace the path here with what is output on your system from the first part of the output of the `whereis llvm-symbolizer` command.
4. Now, run the command

   ```
   source ~/.zshrc
   ```

5. Now, whenever you compile / link programs with the options you uncommented in the `Makefile`, you will see extra output after running them indicating if there are any memory leaks / errors. If there are no memory leaks / errors, the program will run as normal.

Run the `shop` program, either with `valgrind`, or as specified above. Is any memory lost? The answer should be yes — there were a couple of bugs intentionally added in your starter code. Find and fix these bugs, then confirm that no memory is being lost on the functions that you wrote. Hint for debugging: check the places where memory is allocated or deleted. Remember, every new should have a delete.

Part 4: Submitting Your Lab

You need to submit all code files (even those that you did not modify). In addition, from this assignment onwards you also have to provide a README file (reference the style guide for what should be included in your README, lab README's can be brief). Submit your work according to the discussion in class.

A Note on Lab Grading

Labs are for hands-on practice in a supervised setting so you can develop your skill. They are designed to be low-pressure and fun. Come to lab, do your best, submit your work at the end of the period (your best effort given the time), and you will get a high score. If for some reason you miss your lab session you can go to another one (provided there is space, see the course syllabus). If you cannot attend any session for a whole week do it in your own so you will gain the skills. See the syllabus for course policies on labs.