Program 1: Building Solar Systems

CS240 Fall 19

Due Date

Midnight, October 7th, 2019

All programs will be tested on the machines in the G7 lab. If your code does not run on the system in this lab, it is considered non-functioning EVEN IF IT RUNS ON YOUR PERSONAL COMPUTER. Always check that your code runs on the lab machines before submitting.

Driver Code and Test Files

- program1.cpp
- README.md

Grading Rubric

TOTAL: 45 points

- Part A (*10 points) : Vectors*
  - [2 pts] Test 1 Passed: creates a vector with properly initialized values
  - [2 pts] Test 2-3 Passed: inserting into the vector passes all tests
  - [2 pts] Test 4-6 Passed: reading from vector passes all tests
  - [2 pts] Test 7 Passed: removing from vector passes tests
  - [2 pts] Test 8 Passed: random removal
- Part B (*10 points) Lists*
  - [2 pts] Test 9 Passed: creates a list with properly initialized values
  - [2 pts] Test 10 Passed: inserting into list at index passes test
  - [2 pts] Test 11 Passed: reading from list at index passes test
  - [4 pts] Test 12 Passed: removing and deleting from list at index passes test
- Part C (*20 points*): Stars
  - [12 pts] Test 13 Passed: create, read from Starvector and Starlist without error
  - [8 pts] Test 14 Passed: delete Starvector and Starlist without error
- Part D (*5 points*): Profile
  - [5 pts] completes profiling without error
- Part D: Submission
  - Repository includes .o files or binary [-1 point]
  - Memory leak or Memory error [-10 points]
  - Does not follow style guidelines[-3 points]
  - Makefile doesn't have required targets and flags [-3 points]
  - Your submission will not be accepted if:
    - README.md is not completed
    - Does not follow requested project structure and submission format
    - Does not compile

Guidelines and Policies

Debugging

Most labs are set up to allow you to skip specific tests. To select tests to skip, you should set a preprocessor define directive in your makefile, using the -D flag. This sets the preprocessing variable in your makefile *when you compile to an object file*. Below is an example of skipping tests 3-5:

```
g++ -g -Wall -Werror -std=c++14 -c lab.cpp -o lab.o -DTEST3 -DTEST4 -DTEST5
```

Getting Help

Please follow the debugging guidelines outlined here. We will try to answer questions and provide help within 24 hours of your request. If you do not receive a response in 24 hours, please send the request again.

Although we will answer questions, provide clarification, and give general help where possible up until the deadline, we will not help you debug specific code within 24 hours of the deadline. We will not provide any help after the deadline.

Guidelines

This is a pair programming assignment. You and a partner can divide up the work. Although both of you may not work on all parts of the program you should understand and be able to fully explain every portion of the code. Outside of your team, it is permissible to consult with classmates to ask general questions about the assignment, to help discover and fix specific bugs, and to talk about high level approaches in general terms. It is not permissible to give or receive answers or solution details from fellow students.

You may research online for additional resources; however, you may not use code that was written specifically to solve the problem you have been given, and you may not have anyone else help you or your partner write the code or solve the problem. You may use code snippets found online, providing that they are appropriately and clearly cited, within your submitted code.

If you or your partner are found to have plagiarized any part of the assignment, both will receive a 0 and be reported.

By submitting this assignment, you agree that you have followed the above guidelines regarding collaboration and research.

In this lab, you will learn to:

- Work with aggregate sequential data structures

For the first program you and partner are going to build both a dynamic array (vector) and a linked list. Each data structure will be tested to ensure the validity of its operations. Once that is complete and you know your data structures are working, we are going to use them as internal data structures for our Star class. Both versions of the Star class will be evaluated to see which provides better performance.

You will reuse your Planet class from lab 2 with minimal changes.

For all Data Structures, you may (*and probably should*) add additional functions, methods, and attributes, but what follows is the required minimum interface

You may **not** use the STL for this program.

You will need to make a small change to your planet class constructor.

Planet

- `Planet(unsigned int distance)`
  - You do not need to pass an ID. The ID will now be the address of the object.
- `unsigned long getID()`
  - The return type of the getID() method will need to be changed to a long unsigned to hold the address (8 bytes)

You will need to update your `Star` class method `addPlanet` accordingly.

Part A

Vector

You must break up your code into `Vector.h` and `Vector.cpp` according to the conventions we have discussed in class. Create a dynamic array data class, Vector. You must create your internal array on the heap (using `new`). Your Vector class should have, at minimum, the following public interface:

- `Vector()`
  - initializes an empty vector
- `~Vector()`

    - A destructor to clean up memory
- `void insert(unsigned int index, Planet * p)`
    - inserts an element at index, increasing the Vector size by 1
    - if the insert index is out of bounds, you should increase the capacity to the size of the index + 1
- `Planet* read(unsigned int index)`
    - returns a pointer to the `Planet` object at `index`
    - if the index is out of bounds or unused, return `NULL`
- `bool remove(unsigned int index)`
    - remove the `Planet` object at `index`, decreasing the size of the `Vector` by 1
    - if the index is out of bounds, return false.
- `unsigned int size()`
    - returns the current size of the `Vector` (*this may not be the same as the number of elements*)

Part B

Linked Lists

You must break up your code into `List.h` and `List.cpp` according to the conventions we discussed in class. Create a doubly linked list using a `List` and `Node` classes.

Your List and Node classes can go in the same file.

Your linked list should have the following public interface:

- `List()`
    - A pointer to a head and tail node, both initialized to `NULL`
- `~List()`
    - A destructor to clean up memory
- `void insert(unsigned int index, Planet * p)`
    - inserts an element at `index`, increasing the `List` size by 1
    - if the insert index is out of bounds, you should append to the end of the list
- `Planet* read(unsigned int index)`
    - returns a pointer to the `Planet` object at `index`
    - if `index` is out of bounds, return `NULL`
- `bool remove(unsigned int index)`
    - remove the `Planet` object at `index`, decreasing the size of the `Vector` by 1.
    - return `true` on successful deletion or `false` if `index` is out of bounds
- `unsigned int size()`
    - returns the current size of the List

You will need to have parts A and B complete next lab (9/24)

--END OF IN LAB REQUIRED WORK--
You may continue to work on the remainder of the lab on your own time or in lab

Part C

Full of Stars

Although you should have most of the code written from previous labs. You will need to make slight alterations to your Star class to work with the changes to the Planet class and to work with one of the Data Structures (`Vector, List`) you have created. You must break up your code into `Star.h` and `Star.cpp` according to the conventions we discussed in class.

You will have two Star classes, `Starlist` and `Starvector`. You will be using your `List` and `Vector`, respectively, as the internal data structure to hold Planets.

Starlist

Your `Starlist` must have the following public interface:

- `Starlist()`
  - Initialize memory
- `~Starlist()`
  - deallocate all memory when the `Starlist` object is deleted.
- `unsigned long addPlanet()`
  - return the ID of the newly created `Planet` object
- `bool removePlanet(unsigned long)`
  - Takes a Planet's ID as a parameter, and removes the Planet from the Starlist.
  - You must return `true` upon successful deletion and `false` on failure if the ID isn't found.
- `Planet * getPlanet(unsigned long)`
  - Takes a Planet's ID and returns a pointer to the Planet. If the Planet is not found, it returns `NULL`.
- `void orbit()`
  - Iterate through your planets and alter their orbit position by +1
- `void printStarInfo()`
  - Prints the Starlist information.
- `unsigned int getCurrentNumPlanets()`
  - returns the current number of planets stored

Starvector

Your `Starvector` must have the following:

- `Starvector()`
  - Initialize memory
- `~Starvector()`
  - deallocate all memory when the `Starvector` object is deleted.
- `unsigned long addPlanet()`
  - return the ID of the newly created `Planet` object
- `bool removePlanet(unsigned long)`
  - Takes a Planet's ID as a parameter, and removes the Planet from the Star.
  - You must return `true` upon successful deletion and `false` on failure if the ID isn't found.
- `Planet * getPlanet(unsigned long)`
  - Takes a Planet's ID and returns a pointer to the Planet. If the Planet is not found, it returns `NULL`.
- `void orbit()`
  - Iterate through your planets and alter their orbit position by +1
- `void printStarInfo()`
  - Prints the Star information.
- `unsigned int getCurrentNumPlanets()`
  - returns the current number of planets stored (the size of the vector)

Part D

Profiling

Welcome to the easy part. If you made it this far, you can sit back and relax. I have written some profiling that test the operations for both versions of your Star class.

Pay attention to the output.

- What implementation is better for insertion?
- Reading?
- Overall?

You do not need to get any specific numbers, but I encourage you to compare your numbers with others. How is your implementation affecting you execution time?

Part E

Submission

Required code organization:

- program1.cpp
- List.h/.cpp
- Vector.h/.cpp
- Star.h/.cpp
- Planet.h/.cpp
- makefile
    - **You must have the following targets and compilation flags in your makefile:**
        - *FLAGS*: `-g, -Wall, -Werror`
        - all - compiles your source code to an executable called, "**program1**" using separate compilation for each .cpp file
        - clean - removes all object files and binary executables
        - run - compiles if necessary and runs your executable
        - memcheck - compiles your source if necessary, then runs your executable with valgrind
- README.md
    - You must complete the README.md file before submitting.

Before you submit on MyCourses, complete the following form:
https://forms.gle/bCYVeQ1e1BZKbQzJA
Completing this form will allow us to grade your Program 1. You need only submit for one person in your group on MyCourses.

Below is just a reminder of the commands you should use to submit your code. If you cannot remember the exact process, please review Lab 0.

These commands all presume that your present working directory is within the directory tracked by git.

You will need to do the following when your submission is ready for grading.

```
git commit -a -m "final commit"
git push
```

To complete your submission, you must copy and paste the commit hash into MyCourses. Go to MyCourses, select CS240, and then assignments. Select this lab, and where it says **text submission (not comments)**, paste your commit hash. **DO NOT PASTE ANYTHING OTHER THAN YOUR COMMIT HASH.**
Incorrect: "commit hash: 690fa67ed8".
Correct: 690fa67ed8

You can get your latest commit hash with the following command:

```
git rev-parse HEAD
```

Remember, you **MUST** make a submission on mycourses before the deadline to be considered on time.