Project #01: myvector class

Complete By: Saturday, September 7th @ 11:59pm
Assignment: myvector class with alternative implementation
Policy: Individual work only, late work *is* accepted (see "Policy" section for more details)
Submission: zybooks + gradescope

Assignment

In class we have been discussing the implementation of the built-in C++ **std::vector** class, which is based on a dynamically-allocated array. In this project you're going to re-implement vector as your own **myvector** class, using an implementation of your own design.

The assignment consists of 4 parts, which walks you through the process of starting with a simple myvector of integers, and ending with a templated-version of myvector based on your own implementation. Approach the assignment part by part.

Part 01:

See zybooks section 1.9. Complete exercise.

Part 02:

See zybooks section 1.10. Complete exercise.

Part 03:

See zybooks section 1.11. Complete exercise.

Part 04:

At this point, you should have a working **myvector<T>** class that is (a) templated, and (b) implemented in some approach other than the dynamically-allocated array we have discussed in class. Make sure you have 3 constructors: default, copy constructor, and one that takes an initial_size. As discussed in part 03, the goal is for you to develop an approach that improves upon the array-based implementation in some way. For example, perhaps you want to take an approach that avoids the cost of re-allocating the array when it's full, e.g. using a linked-list. Of course, with a linked-list the cost of accessing an element will jump from O(1) to

O(N), so that's a costly trade-off. Perhaps you can think of a better approach that avoids the cost of re-allocating the array, while retaining O(1) access to any element *in most cases*? Take some time to think about your implementation (and the trade-offs), draw some diagrams, and then implement.

Here in part #04, the expectation is that you'll work outside of zybooks so that you can do more extensive testing, since the "Movie Reviews" program is not an extensive test of myvector<T> functionality. How you do this testing we're going to leave to you, but testing is a skill that takes practice to develop. In short, think of ways to break your myvector<T> class, and write code to make sure your vector<T> class works as required. Can you insert thousands

Here in part #04 you are also required to add three additional functions to your myvector<T> class, as defined below:

```
//
// erase
//
// erase the element at position i from the vector by overwriting it and
// reducing the size (# of elements); the ith element is returned
//
// Assume: 0 <= i < Size
//
T erase(int i);

//
// []
//
// returns a reference to the element at position i so the element can
// be read or written
//
// Assume: 0 <= i < Size
//
T& operator[](int i);

//
// rangeof
//
// Returns the elements in the range of positions i..j, inclusive.
// The elements are returned in a dynamically-allocated C-style array.
//
// Example: if i is 2 and j is 4, then an array containing 3
// elements is returned:  element position 2, element position 3, and
// element position 4.
//
// Assume: 0 <= i <= j < Size
//
T* rangeof(int i, int j)
```

In general, assume all parameters to the myvector<T> class are valid --- we'll worry about proper error checking later. Do not add a destructor, we'll discuss destructors and proper memory deallocation in future projects. However, you should free memory within your other functions, e.g. in class we discussed the freeing of the old array in push_back.

Requirements

Your approach needs to use pointers in some way, above and beyond a single pointer to a dynamically-allocated array. A simple one or two-way linked-list is a valid approach, but will not receive full credit --- you can do better than that. Do not use any of the built-in containers of C++ (e.g. do not use std::vector to implement myvector). We aren't looking for the best possible data structure, those are coming later. Just something better than a linked-list, if posible.

You will be expected to update the comments in "myvector.h", including the header comment to explain your implementation approach. Ultimately we will be grading both the correctness and the quality of your implementation in "myvector.h". Be sure the header comment includes your name, and motivates your approach --- what trade-offs are you making in comparison to the dynamic, array-based approach?

Keep all implementation details private. You'll probably need to define a private struct or class, e.g. to define a **NODE** in a linked-list. The proper way to do this is within the *private* section of your **myvector** class, like this:

```
template<typename T>
class myvector
{
private:
  struct NODE
  {
    …
  };

  // rest of myvector class, which can now use struct NODE
};
```

Programming Environment

You are welcome to program on whatever platform you want, using whatever compiler / programming environment you want. If you not sure what to use, or have no environment available, use Codio; see Appendix at end of this document for getting started with Codio.

Have a question? Use Piazza, not email

As discussed in the syllabus, questions must be posted to our course Piazza site — questions via email will be ignored. Remember the guidelines for using Piazza:

1. *Look before you post* — the main advantage of Piazza is that common questions are already answered, so search for an existing answer before you post a question. Posts are categorized to help you search, e.g. "HW".

2. Post publicly — only post privately when asked by the staff, or when it's absolutely necessary (e.g. the question is of a personal nature). Private posts defeat the purpose of piazza, which is answering questions to the benefit of everyone.
3. Ask pointed questions — do not post a big chunk of code and then ask "help, please fix this". Staff and other students are willing to help, but we aren't going to type in that chunk of code to find the error. You need to narrow down the problem, and ask a pointed question, e.g. "on the 3rd line I get this error, I don't understand what that means…".
4. Post a screenshot — sometimes a picture captures the essence of your question better than text. Piazza allows the posting of images, so don't hesitate to take a screenshot and post; see http://www.take-a-screenshot.org/ .
5. Don't post your entire answer / code — if you do, you just gave away the answer to the ENTIRE CLASS. Sometimes you will need to post code when asking a question --- in that case post only the fragment that denotes your question, and omit whatever details you can. If you must post the entire code, then do so privately --- there's an option to create a private post ("visible to staff only").

Electronic Submission and Grading

There are 4 parts to this assignment. Parts 1-3 are worth 40/100, and are graded via zybooks (sections 1.9, 1.10, and 1.11). Part 4 is worth 60/100, and is graded via Gradescope. So the maximum correctness score is the sum of parts 1-4, which is 100/100. Then we'll manually grade for (a) style / commenting, and (b) quality of implementation. For (a), we are looking for readable code (whitespace, indentation, etc.), as well as comments in the header and above each function that explain your data structure; this is worth 10/100 points. For (b), we are looking for an implementation above and beyond a simple one-way or two-way linked-list; this is worth 10/100 points.

Final (part 04) submissions will be collected and graded using Gradescope; you should have received an email to register via your UIC email (do not change this email by the way, it breaks the linkage between Gradescope and Blackboard). If you did not receive an email about Gradescope, you can self-register at www.gradescope.com using the entry code **M2DYJR**. Gradescope submissions will not be accepted until 1-2 days before the due date; you are responsible for your own testing.

When you are ready to submit for grading, login to Gradescope and drag-drop your "myvector.h" file under "Project 01"; if you are working in Codio, you can export your work from Codio (Project menu, "Export as Zip") and then submit the .zip file. Gradescope will run a set of test cases and report your score. You have unlimited submissions, and Gradescope keeps a complete history of all submissions you have made. By default it records the score of your last submission, but if that score is lower, you can click on "Submission history", select an earlier score, and click "Activate" to select it. The activated submission will be the score that gets recorded, and the submission we grade. If you submit on-time and late, we'll grade the last submission (the late one) unless you activate an earlier submission.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your “iClicker teammates”), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else’s work (partial or complete) as your own. The University’s policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else’s iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .

Appendix: Codio Programming Environment

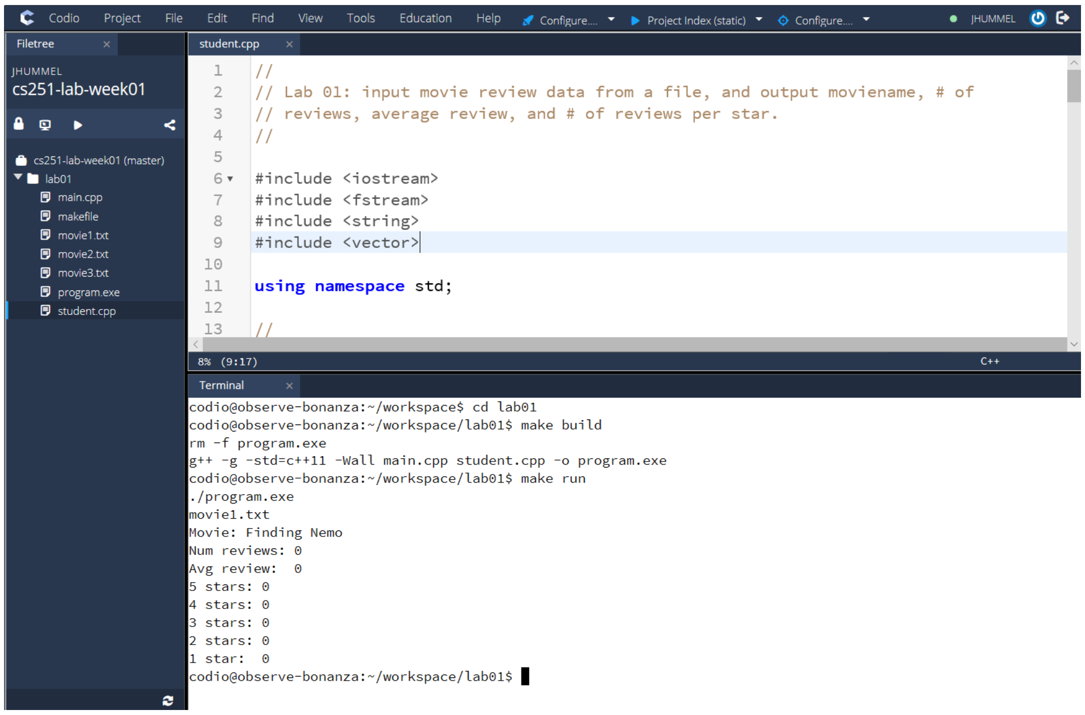
Here's a quick summary of how to get started with Codio. Codio is cloud-based, and accessible via a web browser. It's platform-neutral and works from any platform, but you must be online to use it. The first step is to create a Codio account and join the class ("CS 251 Fall 2019"):

https://codio.com/p/join-class?token=brigade-america

Be sure to register using your UIC email address, especially since you may be using this account in future CS classes. The above link will provide access to Codio, and the resources associated with CS 251.

Once you successfully login to Codio, you'll see the project "**cs251-project01-myvector**" pinned to the top of your dashboard. This represents a container-based C++ programming environment --- think light-weight virtual machine (VM). When you are ready to start programming, click "Ready to go" and Codio will startup the VM and within a few seconds you'll have a complete Ubuntu environment at your disposal. In particular, you'll have access to C++ via the GNU g++ compiler. You also have super-user (root) access, so you can install additional software if you want ("sudo apt-get install XYZ").

Once I login, I normally split the right-side into 2 windows: the top as my editor pane, and the bottom as my terminal window. This can be done via the View menu, Panels, Split Horizontal. Then click on the bottom window, drop the Tools menu, and select Terminal. Here's a snapshot showing "student.cpp" (from lab 01) open in the editor, and the terminal window open below it:



```
//
// Lab 01: input movie review data from a file, and output moviename, # of
// reviews, average review, and # of reviews per star.
//

#include <iostream>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

//
```

```
codio@observe-bonanza:~/workspace$ cd lab01
codio@observe-bonanza:~/workspace/lab01$ make build
rm -f program.exe
g++ -g -std=c++11 -Wall main.cpp student.cpp -o program.exe
codio@observe-bonanza:~/workspace/lab01$ make run
./program.exe
movie1.txt
Movie: Finding Nemo
Num reviews: 0
Avg review:  0
5 stars: 0
4 stars: 0
3 stars: 0
2 stars: 0
1 star:  0
codio@observe-bonanza:~/workspace/lab01$
```

For the purposes of project #01, nothing is provided other than a makefile --- use the File "Upload…" to upload your "myvector.h" code from zybooks, along with other files you might need. At the very least you'll want to create a new "main.cpp" file for testing purposes, and #include "myvector.h". You can use the provided makefile to compile and run: type "make build" to compile, and "make run" to execute.