CSCI-1200 Data Structures — Spring 2021
Lab 1 — C++ Development, STL Strings, & STL Vectors

Welcome to CSCI 1200 Data Structures lab! Please check your email for your assigned lab study group. We hope to have this information distributed by Wednesday January 27th, at 10am. You should have a WebEx Meeting URL, the names of your classmates assigned to your study group, and the name of a *graduate student teaching assistant (TA)* who is supervising your lab time period. The graduate TAs will be joined by a team of *undergraduate student programming mentors* to manage the weekly Wednesday labs. **Your TAs and mentors are eager to answer any questions about the course materials and work with you one-on-one to master strong programming and debugging skills.**

Please join the WebEx meeting at the specified start time. Our plan is to keep the study group assignments fixed for the duration of the semester. So, introduce yourself to your study group classmates and ask questions to find out a little bit about each other. Practice *sharing your screen* and different application windows with the group – you will do this to ask for help on the lab exercises, to assist your study group members if they get stuck, and to demonstrate your completion of the lab exercise to the TA.

You should plan to spend the full 1 hour and 50 minute session connected to the WebEx call each week. Some weeks you may finish the lab exercises early and be able to leave early – but do not expect this to happen every week. **Please turn on your audio and video, and leave them enabled for the duration of the call.** You may mute the audio when you aren't talking if you have unavoidable background noise. If necessary (if your internet access is slow or inconsistent), you may need to temporarily disable the video to improve the audio quality – but please try to keep the video on as much as possible.

There will be three graded exercises or "checkpoints" associated with each lab. You are encouraged to talk with your study group classmates about the lecture material, the lab exercises, and about C++ programming skills. This will help reduce the burden on the TAs and will reduce your waiting time in lab. **Note: Each student must produce his/her own exercise solutions.** To earn credit for each checkpoint you will need to answer short questions about the material. If you have done the checkpoint and understood it, you should have no trouble earning this credit. If you have relied on help from other students too much, you may find the questions hard to answer.

Your TA and mentors will be jumping in and out of the different lab study group WebEx Meetings that are happening during your time block. They may drop without invitation, or you may request that someone come help or check you off by placing your study group name and contact information in the office hours queue:

https://submitty.cs.rpi.edu/courses/s21/csci1200/office_hours_queue

Do not wait until the end of lab to be checked off for multiple checkpoints. If the queue is sizeable, the TA/mentor will only check you off for one checkpoint at a time and ask you to add your name to the end of the queue for the next checkpoint. Each lab period is 1 hour and 50 minutes long. The office hour queue for help and checkpoints may close up to 10 minutes before the end of the lab period. Barring extenuating circumstances, no checkpoints may be earned outside of the lab period.

Today we focus on using the terminal command line and g++ to compile, run, and inspect the results of your program. After today's lab you are welcome to explore other options for your C++ development environment. However, for the homework assignments, your code must compile and run correctly under gcc/g++ 7.5 and/or llvm/clang++ 6.0 on Ubuntu 18.04. This streamlined grading process allows the TAs to spend more time giving you constructive feedback on programming style, individual tutoring, and debugging help.

Checkpoint 1

estimate: 30 minutes (+ installation delays??)

- The course website includes instructions to install and setup the necessary software for Windows, MacOSX, and GNU/Linux. Windows users will need Windows 10 and Windows Subsystem for Linux (WSL) to follow the instructions below. Ask your TAs and mentors for help if you get stuck.
  http://www.cs.rpi.edu/academics/courses/spring21/csci1200/development_environment.php

- **Create a directory (a.k.a. "folder")** on your laptop to hold Data Structures files. Create a sub-directory to hold the labs. And finally, create a sub-directory named `lab1`. Please make sure to save your work frequently and periodically back-up all of your data.

- Using a web browser, copy the following files to your `lab1` directory:
  http://www.cs.rpi.edu/academics/courses/spring21/csci1200/labs/01_getting_started/quadratic.cpp
  http://www.cs.rpi.edu/academics/courses/spring21/csci1200/labs/01_getting_started/README.txt

- **Open a shell/terminal/command prompt window**. *Please ask for help if you have problems installing WSL or finding your* `bash` *shell or terminal.*

How to use the Terminal Command Line: Typical Structure

command	arguments(s)	option	argument for option	another option
↓	↓	↓	↓	↓
`g++`	`main.cpp`	`-o`	`test.exe`	`-Wall`

Each *command* will typically be structured somewhat like the one above. First comes the name of the command, like "`g++`" or "`ls`" or "`cd`". Then come any *arguments* that the command takes (some commands don't take any – some take a lot). The command may also have *options*, like "`-l`" for the "`ls`" command, which displays the *long* format listing with dates and sizes, etc. Options can have arguments as well, like the "`-o`" command for "`g++`", which expects a name for the executable that will be created. Display a help message for the command by typing the command name and then "`--help`". You can learn about a command's options by typing "`man`" and then the command name to view its manual page. Google is also a helpful resource for learning about command options.

Listing Files

`ls` *or* `ls Documents/RPI/DS/Homeworks`

List the files in a directory with the ls command. You can just type "`ls`" for the current directory, or "`ls`" and then a path to a directory to view that directory's contents. If you need to view more detailed information about each file (like the date modified, file size, permissions, etc.), use "`ls -l`".

Changing directories

`cd lab1` *or* `cd ../../homeworks/hw1`

Change directories with the "`cd`" command. You can navigate to an immediate subdirectory by specifying just that subdirectory name, or you can jump several levels away separating each directory name with a "`/`". Use "`cd ..`" to go up a level to the parent directory. You may specify an *absolute path* by starting with the top level *root* directory "`/`"; otherwise it is a *relative path* starting at the current directory. Note: "`./`" refers to the current directory and "`~/`" is your *home* directory. On Windows/Cygwin, to get to the `C` drive you will type something like "`cd /cygdrive/c`".

- **Within the terminal, navigate to** your Data Structures Lab 1 directory and **inspect the contents of your file system** as you go using the "`ls`", "`cd`", and "`pwd`" commands.

  In doing so, remember that directory names are separated by a forward slash "`/`" and when you have a space in the name of the directory, you precede the blank with a backslash "\". Thus, you may type something like this:

```
cd /Users/username/My\ Documents/Data\ Structures/labs/lab1
```

- Confirm that the files `quadratic.cpp` and `README.txt` are in the *current* directory (use `ls`).

Where am I? `pwd`

Use this command to *print* the (current) *working directory*.

Auto-complete - *just hit tab*

You can use the tab key to auto-complete a command, directory, or filename after typing the first few letters (if the completion is unique).

- First, let's confirm that gcc is installed on your machine and check the version by typing:

  ```
  g++ -v
  ```

  If you are not using Ubuntu 18.04 and gcc/g++ 7.5 or clang/clang++ 6.0, you *may* notice slight differences between your compiler and the version on the homework submission server. But don't worry if you have a different version! We will primarily be using parts of C++ that have been stable and unchanged for many years. You may also try to compile using `clang++` instead of `g++`. The LLVM/clang++ compiler has earned much praise for having clear and concise compiler error messages that are especially helpful for new C++ programmers. Note that on MacOSX `g++` is probably actually *aliased* to run `clang++` instead. This is not a problem!

- Now you are ready to attempt to **compile/build the program** for this lab by typing:

  ```
  g++ quadratic.cpp -o quadratic.exe -Wall
  ```

Compilation `g++ main.cpp my_class.cpp -Wall` *or* `g++ *.cpp -o test.exe`

After the compiler name ("`g++`" or "`clang++`"), list all of the `.cpp` files that you want to be compiled (later when we use `.h` files, you will **NOT** list them for the compiler, they will be `#include`-d instead). You can manually list out the files or, if you want to specify all of the `.cpp` files in the current directory, just use "`*.cpp`". The "`*`" searches for all files that match that pattern.

The process of *compiling* a program translates the high-level C++ code into machine-level, "object" code, which is then *linked* with pre-compiled libraries to produce an *executable*. You can specify the name of the executable with the "`-o`" option (or it will name your program "`a.out`" on GNU Linux/OSX or "`a.exe`" on Windows by default).

If the compiler gets confused by a problem with your code and cannot create an executable, it will print out *error* messages. You must correct all errors before you can run the program.

In addition to errors, the compiler may find lines of your code that look suspicious. If possible, the compiler will report these issues as *warnings*, but still produce an executable you may run. You should look closely at all warnings (they may be problematic bugs in your logic!) and it is good practice to correct these issues as well. We recommend using the "`-Wall`" option to compile with *all warnings enabled*.

We have intentionally left a number of errors in this program so that it will *not* compile correctly to produce an executable program. *Don't fix them yet!*

"Submit" the buggy version of the lab code to Submitty:
http://www.cs.rpi.edu/academics/courses/spring21/csci1200/homework_policies.php
Follow the instructions to submit the `quadratic.cpp` and `README.txt` files to Lab 1. After submitting the buggy code you should receive confirmation of your submission and be notified of the compile-time errors in the program. Note that all homeworks will require submission of both your code and README.txt file to receive full credit.

To complete Checkpoint 1: Show one of the TAs the compiler errors that you obtained in the g++/clang++ development environment on your machine *and* the response from the homework submission server indicating the same compiler errors.

Checkpoint 2

estimate: 15-30 minutes

Using previous commands - up/down arrows, `history`, and `!`

You can use the up and down arrows of the keyboard to navigate through old commands so you don't have to retype them. Type "`history`" to view a list of recently run commands. For example, if you had just run a "`g++`" command, made some file edits and wanted to re-compile, you could press the up arrow and the "`g++`" command would show up as if you had just typed it. Use the "`!`" command to search the recent command history and re-run commands. "`!!`" will re-run the previous command (same as typing up arrow, then enter). If you want to go back 2 commands, use "`!-2`". You can also search using "`!`" and then a string. If you ran "`!g++`", it would find the most recent command starting with "`g++`", like "`g++ main.cpp -Wall -o test.exe`", and re-run it. These tricks are very useful so you don't have to painstakingly retype commands!

The compiler errors we have introduced are pretty simple to fix. Please do so, and then re-compile the program on your own machine. Once you have removed all of the errors, you are ready to execute the program by typing:

```
./quadratic.exe
```

After testing the program on your own machine with a variety of inputs, and convincing yourself everything looks good, then you can **"Re-submit" the fixed version of the lab code to the homework server**. Assuming your fixes are cross-platform compatible, the re-submission should successfully compile and run without error. Note that Submitty allows you to review the autograding results of all prior submissions.

Showing a text file - `cat`, `less`, `head`, and/or `tail`

These commands can be used to print the contents of a code or plaintext file on the screen. This is useful for checking any program output written to a text file. `cat` displays the whole file (it may scroll off the screen), `less` shows one page of the file at a time (use space bar to see the next page), `head` shows the first lines of the file, and `tail` shows the last lines of the file.

To complete Checkpoint 2: Show the TA the results of submitting your debugged code for the one equation program to the homework server *and* your debugged, extended, and tested multi-equation program (not submitted to the server).

Checkpoint 3

Checkpoint 3 will be available at the start of Wednesday's lab.