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

Welcome to CSCI 1200 Data Structures lab! Please listen carefully when your graduate lab TA and undergraduate programming mentors introduce themselves at the start of class. They are here to answer any questions about the course materials and work with you one-on-one to master strong programming and debugging skills. Also, introduce yourself to the other students in your lab section.

There will be three graded exercises or "checkpoints" associated with each lab. You are encouraged to talk with your 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.

If you have a question about the exercise or if you are ready to be checked off, add your name to the appropriate Submitty Office Hours Queue for the next available TA/mentor:

https://submitty.cs.rpi.edu/courses/f21/csci1200/office_hours_queue

Do not wait until the end of lab to be checked off for multiple checkpoints. If other students are waiting in the queue 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. Class ends 10 minutes before the hour and no checkpoints may be earned after this time. Barring extenuating circumstances, no checkpoints may be earned outside of the lab period.

You should plan to spend the full 1 hour and 50 minutes in the lab room every week. If you do finish your lab work early, you can ask the TA and mentors questions about this week's Data Structures Homework or about the upcoming exam. Some weeks you may finish the lab exercises early and be able to leave early – but do not expect this to happen every week.

> *IMPORTANT NOTE: No phones, no email, no texting, no social media, no web surfing, no game-playing, no distraction! With the exception of downloading lab files provided by the instructor at the start of lab, and occasional use of online C++ reference material (e.g., to look up the the details of a particular built-in function or class), you are not allowed to use the internet during lab. Anyone caught using their cell phones, the internet, or programs not directly relevant to this course will be given an immediate 0 for that lab and asked to leave.*

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++ 9.3 and/or llvm/clang++ 10.0 on Ubuntu 20.04. 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 the 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/fall21/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/fall21/csci1200/labs/01_strings_vectors/quadratic.cpp
  http://www.cs.rpi.edu/academics/courses/fall21/csci1200/labs/01_strings_vectors/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/WSL, to get to your home directory the `C` drive you will type something like "`cd /mnt/C/Users/`".

- **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 20.04 and gcc/g++ 9.3 or clang/clang++ 10.0, 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:
https://submitty.cs.rpi.edu/courses/f21/csci1200/
Upload the `quadratic.cpp` and `README.txt` files to Lab 1 Practice gradeable. 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.

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 1: Show a TA or mentor the compiler errors that you obtained in the g++/clang++ development environment on your machine *and* the response from Submitty indicating the same compiler errors. Also show the edits you made to the code to fix these problems both on your machine and on Submitty.

Checkpoint 2

estimate: 30 minutes

Now let's write a brand new C++ program to learn about command line arguments. First open up a brand new file named `silly.cpp`. Include `<iostream>` at the top of the file.

Read the reference sections of the course webpage explaining command line arguments in C++:
`http://www.cs.rpi.edu/academics/courses/fall21/csci1200/programming_information.php`

- To start, let's write a program that expects only integers on the command line, and it will print the product (multiplication) of those numbers to the console (`std::cout`).

  Compile and test your program:
  ```
  g++ -Wall -g -o silly.out silly.cpp
  ```
  If we run:
  ```
  ./silly.out 2 3 4
  ```
  Then program will print:
  ```
  product of integers:  24
  ```

And if we run:

```
./silly.out 3 -1 2 20 5
```

Then program will print:

```
product of integers:  -600
```

Hint: Read the section of the course webpage on converting strings to integers.

- Now modify the program to expect strings and to print those strings in alphabetically-sorted order. For example if we run:

  ```
  ./silly.out dog buffalo horse turtle owl
  ```

  Then program will print:

  ```
  sorted strings:  buffalo dog horse owl turtle
  ```

  Hint: Use an STL `vector` *and the* `sort` *function.*

- If you have less than 45 minutes remaining in lab, get checked off now and move on to Checkpoint 3. If you have 60 minutes or more remaining in lab, then let's make your program work with a mix of both integers and strings.

  For example if we run:

  ```
  ./silly.out dog 2 buffalo horse 3 4 turtle owl
  ```

  Then program will print:

  ```
  product of integers:  24
  sorted strings:  buffalo dog horse owl turtle
  ```

To complete Checkpoint 2: Show a TA or mentor your program. Be ready to demonstrate that your program works with other input requested by the TA or mentor.

Checkpoint 3

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