

CSCI-1200 Data Structures — Spring 2016

Lab 1 — Getting Started

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. You may ask your fellow students questions about the lab. 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.**

There will be three graded “checkpoints” associated with each lab. If you have a question or when you have completed each checkpoint, raise your hand or put your name in the appropriate queue and your graduate TA or one of the programming mentors will check your work. Part of earning each checkpoint for the lab will involve answering 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.

Do not wait until the end of lab to be checked off for multiple checkpoints. If there is a 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.

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 details of a particular built-in function or class), you are not allowed to use the internet during lab. Disable your wireless connection if necessary to remove the temptation. 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 will 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++ 4.8.X. This streamlined grading process allows the TAs to spend more time giving you constructive feedback on programming style, individual tutoring, and debugging help.

- Windows users will need Cygwin to follow the instructions below. The default installation of Cygwin from RCS does not include all of the packages you will use this semester. Upgrading is much faster than the original installation. Read more here:
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/development_environment.php
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/OS_choices.php
<http://www.cs.rpi.edu/academics/courses/spring16/csci1200/cygwin.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/spring16/csci1200/labs/01_getting_started/quadratic.cpp
http://www.cs.rpi.edu/academics/courses/spring16/csci1200/labs/01_getting_started/README.txt
- **Open a shell/terminal/command prompt window.** Windows users should be able to find Cygwin in the list of programs. Within the Cygwin directory, you want the `bash shell`. *Please ask for help if you have problems installing Cygwin or finding your Cygwin shell.*

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 g++ 4.8.X, you *may* notice slight differences between your compiler and the version on the homework submission server when we get to advanced topics. 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.

- 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*.

Checkpoint 1

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 the homework server:

<http://www.cs.rpi.edu/academics/courses/spring16/csci1200/homework.php>

Follow the instructions under the “Electronic Submission” section to zip up and 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++` development environment on your machine *and* the response from the homework submission server indicating the same compiled errors. Also, give the TA your signed “Collaboration Policy and Academic Integrity” form (handed out in lecture on Tuesday). Here’s another copy if you need to print it out:

http://www.cs.rpi.edu/academics/courses/spring16/csci1200/csci1200_collaboration_and_academic_integrity.pdf

Checkpoint 2

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

```
./quadratic.exe
```

“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. You will need to show your successful submission to a TA. *But you’re not done yet...*

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!

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.

For the rest of this lab we will work with arrays and the logic of manipulating them. Modify the main program so that instead of a `while` loop, the program will read in 5 different quadratic polynomials from the keyboard. We will assume that all of the a coefficients are 1. So we only need to read in and store the 5 b coefficients into an integer array named `b_array` with 5 slots and similarly read and store 5 c coefficients into a variable `c_array`. Hint: Use a `for` loop. After reading in all of the data, write another `for` loop to find the roots for each of the 5 equations. Store the smaller root for each equation in a third array of 5 spots named `smaller_root`. *Note: You won't submit this modified version of the program to the homework server!*

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

Note that solving Checkpoint 2 did not actually require arrays. We used arrays to get some practice. This final checkpoint, however, requires the use of arrays to store values for later use. For Checkpoint 3, write a new program that reads a sequence of floats into an array, computes and prints the average, and then outputs all of the original numbers (in any order) that are less than the average.

The first value input will be an integer giving the number of values in the sequence. The remaining input will be the values. You may assume there are at most 100 values. You do not need to write a separate function — you may keep all of the code in the main function.

As an example, given the input:

```
3      2.0  1.5  4.0
```

the program should output the the average value 2.5, and also print 2.0 and 1.5 because they are less than 2.5. Given this larger input dataset:

```
15     40.2 9.4 -2.5 26.1 37.1 44.3 56.8 11.5 17.8 39.9 42.2 -1.7 89.2 65.1 27.1
```

the program should output the average 33.5 and also print the 7 values greater than the average: 9.4 -2.5 26.1 11.5 17.8 -1.7 27.1

To complete Checkpoint 3 (and the entire lab): show the resulting program and its output to a TA.