

Lab 1

Due Sep 15, 2023 by 6:30p.m.

Points 1



Important: You must not commit and push executable files to your repository. First, they will break the tests and you will get 0 on the lab. Second, you will have learned in CSC207 that automatically generated files do not belong in repositories. For this lab, your repository should contain the two shell scripts that are required, and the provided starter code. It should not contain the zip file or any executables. It may contain text files that include notes that you want to keep track of with your lab. You can use `git rm <file name>` followed by `git commit` and `git push` to remove these files from your repo.

Lab 1: Introduction to the Unix Shell and C Programs

Introduction

The purpose of this lab is to practice using a few different shell commands to navigate through the file system, review git, and compile and run simple C programs.

Before starting this lab, we strongly recommend you complete the following sections on the PCRS:

- C Language Basics -> DISCOVER: [Types, Variables and Assignment Statements](https://pcrs.teach.cs.toronto.edu/csc209-2023-09/content/challenges/7/1)  (<https://pcrs.teach.cs.toronto.edu/csc209-2023-09/content/challenges/7/1>) (video 1)
- C Language Basics -> DISCOVER: [Input, Output and Compiling](https://pcrs.teach.cs.toronto.edu/csc209-2023-09/content/challenges/101/1)  (<https://pcrs.teach.cs.toronto.edu/csc209-2023-09/content/challenges/101/1>) (all videos)

You'll find the other videos in the "C Language Basics" part useful as a reference as you start working with C.

You should also have your computing environment set up (see the [Software Setup](https://q.utoronto.ca/courses/309775/pages/software-setup) (<https://q.utoronto.ca/courses/309775/pages/software-setup>) page on Quercus) before starting this lab.

Learning Outcomes

By the end of this lab students will be able to:

- use git on the command line to clone a repo, make some changes, and commit and push changes
- use basic navigation command line tools to view their file system layout including hidden files
- explain and use standard input and command line arguments as mechanisms for providing input to a program
- compile and execute a C program
- use input and output redirection operators in a shell command
- use the pipe operator in a shell command
- create a trivial shell program

0. MarkUs and git

To start, login to MarkUs and navigate to the `lab1` assignment. You'll find your repository URL to clone. *Even if you know your MarkUs repo URL, it's important to visit the page first since MarkUs needs to create the lab1 folder in the repo.* **Then, at the bottom of the assignment page, click on the button labelled "Add Starter Files to Repository".**

Then, open a terminal on your computer and do a `git clone` of your CSC209 MarkUs repository. Inside the repository, you should see a new folder called `lab1` containing the starter files for this lab.

At the top level of your repo, you will also see a folder called `markus-hooks`. You can safely ignore this folder, but do not delete it. MarkUs expects the git repos that it manages to have a specific structure and it uses git-hooks on the server to enforce this structure. In particular,

- You may not add, delete, or modify top-level files or directories in your repository. Instead, all of your work should be done in assignment-specific sub-directories, like `lab1`.
- When an assignment has required files, you will receive a warning message if your repo is missing some of these files. If the assignment restricts your submission to *only* those required files, you won't be able to commit any other files inside that assignment's subdirectory.

Because these checks are also made by the MarkUs server when you push your code, your submission may not be accepted (the push will fail) if your changes fail to satisfy the above conditions.

1. Basic utilities

Your first task is to inspect the contents of the `lab1` folder. In the command line, use the `ls` command to inspect the contents of this folder. Use the [man page for `ls`](http://man7.org/linux/man-pages/man1/ls.1.html) (<http://man7.org/linux/man-pages/man1/ls.1.html>), which we encourage you to explore the different options you can pass to `ls` as command-line arguments to vary its behaviour. Note that you can pass multiple command-line arguments to `ls`.

Play around with these options now, and see if you can do each of the following: (Note that these commands will show you both files and directories.)

1. Use `-l` to show metadata about each file in the current working directory.
2. Show the same metadata about each file as `-l`, except do *not* show the owner or group of the file.
3. Show only the filename and size of each file, one per line.
4. Show all files in the current working directory including the *hidden* files, which are files that start with a period `.`

2. Compiling and running C programs

You should see a bunch of different C source code files being listed by `ls`. For each one, do the following:

1. Compile it by running the command `gcc -Wall -g -std=gnu99 <filename>`. The arguments `-Wall`, `-g`, and `-std=gnu99` are the standard compiler flags we'll be using in this course – more about these later.
2. Run it according to its usage. Note that some of the executables take no command-line arguments, some require at least one command-line argument of a certain form, and some will read in keyboard input. Try running each program and read the code to learn what each program does!

3. A simple script

Executing commands one at a time in the shell is not scalable: often we have a set of commands we want to execute together repeatedly. We can do so by writing a *shell script*, which is a program written in a shell programming language like `bash`. We'll return to shell programming at the end of this course, but for now, you will write the simplest type of shell program: a list of commands.

To do this, create a new file in your `lab1` directory called `compile_all.sh`; you can do this using any text editor you like. The first line of the file should be the following:

```
#!/usr/bin/env bash
```

This line is called a [shebang](https://en.wikipedia.org/wiki/Shebang_%28Unix%29) (https://en.wikipedia.org/wiki/Shebang_%28Unix%29) line, and is used

to tell the operating system to execute the program following the "`#!`". In this case, it executes the "`bash`" program. The rest of the file is used as input to the "`bash`" program.

In this file, write one line for each compilation command you ran in Part 2. **If you are on Windows, you need to ensure your text editor is using Unix-style line endings (aka "LF" or "\n") for this file.**

Then in the command line, run your script just as you would any other executable:

```
$ ./compile_all.sh
```

This should fail! The operating system will not run this program because the file is not executable. On Unix the permission settings of the file determine whether a file is executable.

To change the permissions we will use the program `chmod`:

```
$ chmod a+x compile_all.sh
```

You can read the man page (`man chmod`) to see what arguments `chmod` accepts. In this case, the 'a' means that we want to change the permissions for all users, the '+' means that we are adding permissions, and the 'x' means the executable permissions.

Now try running the shell script again.

If you inspect the contents of your folder using `ls -l`, you should see that the compilation has been successful and an executable has been produced. Unfortunately, there's a problem: because `gcc` uses the default executable name `a.out` for the executable, each compilation command in your script overwrites the result of the previous step. To fix this problem, modify your script by using the `-o <out_name>` `gcc` flag to produce executables whose names match the C source files, without the extension.

For example, use the command

```
gcc -Wall -g -std=gnu99 -o hello hello.c
```

to compile `hello.c` into an executable named `hello`.

Then when you re-run your script, you should be able to see the resulting executables all created.

Note: this compilation script is a poor way to automate the compilation of multiple C programs in practice. Later in the term, we'll learn about the Unix software tool `make`, which is the industry standard for managing compilation.

4. Redirection, pipes, and more Unix utilities

Now that we have some programs, your next task will be to review how input and output redirection work, and review some basic shell utilities. Create a second shell script called `my_commands.sh` that contains commands that perform each of the following tasks (each of the following should be done in just a single line, and be performed in order):

1. Run `echo_arg` with the command-line argument `csc209` and redirect standard output to the file `echo_out.txt`.
2. Run `echo_stdin` with its standard input redirected from the file `echo_stdin.c`.
3. Use a combination of `count` and the Unix utility program `wc` [↗\(https://linux.die.net/man/1/wc\)](https://linux.die.net/man/1/wc) to determine the *total number of digits* in the decimal representations of the numbers from 0 to 209, inclusive.
4. Use a combination of `echo_stdin` and `ls` [↗\(http://man7.org/linux/man-pages/man1/ls.1.html\)](http://man7.org/linux/man-pages/man1/ls.1.html) to print out the name of the **largest** file in the current directory. You can assume the name of largest file has fewer than 30 characters. Hint: `echo_stdin` prints the first line of standard input, and there is an option to `ls` to print the files ordered by size.

5. Test your shell scripts on teach.cs

Use git to add, commit, and push the two shell programs: `compile_all.sh` and `my_commands.sh` to your repository, making sure they are in the `lab1` folder. **DO NOT** commit the executable files!

Use ssh to log into teach.cs. In a terminal, run the following:

```
ssh USERID@teach.cs.toronto.edu
```

The USERID will be your account id on teach.cs which is most likely the same as your UTORid. The password is likely different than your UTORid password. If you have forgotten your teach.cs password, you can reset it [here](https://www.teach.cs.toronto.edu/resources/cdf_account_management.html) [↗\(https://www.teach.cs.toronto.edu/resources/cdf_account_management.html\)](https://www.teach.cs.toronto.edu/resources/cdf_account_management.html).

Now you have a terminal (shell) running on the teach.cs server! Decide where in your account you want to clone your CSC209 repository, and run `git clone` to get a working copy of your repository in your account on teach.cs.

cd into your lab1 directory (folder) and try running your shell scripts just as you did in steps 3 and 4. Try using an editor like `nano` [↗\(https://linuxize.com/post/how-to-use-nano-text-editor/\)](https://linuxize.com/post/how-to-use-nano-text-editor/), `vim` [↗\(https://www.digitalocean.com/community/tutorials/getting-started-with-vim-an-interactive-guide\)](https://www.digitalocean.com/community/tutorials/getting-started-with-vim-an-interactive-guide), or `emacs` [↗\(https://opensource.com/article/20/3/getting-started-emacs\)](https://opensource.com/article/20/3/getting-started-emacs) to open and edit one of the files. You will need to learn a terminal-based editor to do at least a little bit of work on a remote system like the teach.cs servers and now is a good time to start learning one.

If you found any errors, you can correct them and try again. Remember to use git add, commit, and push to ensure that all changes are in your submission.

Submission

Use git to submit your final `compile_all.sh` and `my_commands.sh` – make sure they're inside your `lab1` folder and named exactly as described in this handout, as that's where our test scripts will be looking for them.

You do *not* need to submit anything for Sections 1, 2, or 5.

IMPORTANT: make sure to review how to use git to submit your work to the MarkUs server; in particular, you need to run `git push`, not just `git commit` and `git add`. Watch carefully for error messages, since committing non-required files may cause the push to fail.