Lab 1

Due Jan 11, 2019 by 6:30pm **Points** 1

Lab 1: Introduction to the Unix Shell and C Programs

Due date: Friday 11 January before 6:30pm. Submissions made after the deadline will not be accepted.

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 (https://pcrs.teach.cs.toronto.edu/csc209-2019-01) :

- C Language Basics -> DISCOVER: Types, Variables and Assignment Statements (video 1)
- C Language Basics -> DISCOVER: Input, Output and Compiling (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 page on Quercus) before starting this lab.

0. MarkUs, git, and git hooks

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!* This triggers the starter code for this lab to be committed to your repository.

Then, open a terminal on your computer and do a `git pull` in your CSC209 MarkUs repository. You should see a new folder called `lab1` with some starter code inside. Make sure to do all your work on this lab in that folder.

You should also see a folder called `markus-hooks`. This semester, we are using a tool to help you make sure you're using git correctly with MarkUs. A *git hook* is a program that runs at a specific time in the git workflow to perform specific checks about the action being run. In this course, you'll be using a *pre-commit hook* to perform two checks every time you commit changes:

- 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 subdirectories, like `lab1`.
- When an assignment has required files, you'll 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. **You can set up git hooks on your computer, so that you will be prevented from committing changes that will violate the checks:**

1. **OSX and Linux users only**: open the file `markus-hooks/pre-commit`. You'll need to modify the command inside the file to use `python3` (or `python3.6`, depending on your setup) rather than just plain `python`, which likely refers to a Python 2 interpreter.
2. In your new cloned repository, copy the file `markus-hooks/pre-commit` into `.git/hooks`:

   ```
   $ cp markus-hooks/pre-commit .git/hooks
   ```

   Note the period at the start of `.git`; this is a *hidden folder*, and may or may not show up in a graphical file explorer, depending on your settings. Running the above command in a terminal is the safest approach.
3. Check your work: run `ls .git/hooks`. One of the files listed should be `pre-commit`.

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)** , 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 "."

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 -std=gnu99 -g <filename>`. The arguments `-Wall`, `-std=gnu99`, and `-g` 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'll 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) line, and is used to tell the operating system to interpret the contents of the file as a shell 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 -std=gnu99 -g -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 final 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 the 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) 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) to print out the name of the **largest** file in the current directory. You can assume the largest file has a name with fewer than 30 characters.

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 or 2.

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.