Lab 1: Introduction to the Unix Shell and C Programs

Due date: Monday, May 14, 2018 before 23:30 pm. 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, compile and run simple C programs, and practice writing C programs involving pointers and the `scanf` function.

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

- C Language Basic -> DISCOVER: Types, Variable and Assignment Statements (Video 1)
- C Language Basic -> 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 Lab 0) before starting this lab.

0. MarkUs and git

To start, login to MarkUs and navigate to the labs 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.

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, 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, -stdgnu99, 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 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` 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` 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.

5. invest.c

Your task is to implement a function `invest` that takes an amount of money and multiplies it by a given rate. It's your job to figure out the exact type of this function's arguments, given the sample usage in the `main` function in the starter code.

6. score_card.c

Your task is to implement a function `sum_card`, which takes an array of pointers to integers, and returns the sum of the integers being pointed to.

7. phone.c

Your task is to write a small C program called `phone.c` that uses `scanf` to read two values from standard input. The first is a 10 character string and the second is an integer. The program takes no command-line arguments.

If the integer is zero, the program prints the full string to standard output followed by a single newline `\n` character. If the integer is between 1 and 9 inclusive the program prints only the corresponding

digit from the string to `stdout` (again followed by a newline). In both of these cases the program returns 0.

If the integer is less than 0 or greater than 9, the program prints the message "ERROR" (followed by a newline) to `stdout` and returns 1.

We haven't learned about strings yet, but you will see that to hold a string of 10 characters, you actually need to allocate space for 11 characters. The extra space is for a special character that indicates the end of the string. Use this line

```
char phone[11];
```

to declare the variable to hold the string. Other than this line, there is no starter code for this program.

You may assume that the user correctly enters a 10-character string and an integer. You must not print anything else to `stdout`. *Do not prompt the user to enter the values*.

Hint: If you skipped the video titled, Input, Output and Compiling Video 3: Reading Input (scanf), now would be a good time to go watch it.

8. phone_loop.c

Your task is to write a C program called `phone_loop.c`. This program will again read from standard input using `scanf` and take no command-line arguments. Similar to `phone.c`, this program reads a 10-character string as the first input value, but then it repeatedly reads integers until standard input is closed. Hint: Use a while loop with condition while (scanf(...) != EOF) rather than ignoring the return value from `scanf`.

After each integer the output produced is as before:

- if the integer is 0, the full string is printed
- if the integer is between 1 and 9, the individual character at that position is printed
- if the integer is less than 0 or greater than 9, the message "ERROR" is printed

In each case the printing is followed by a newline character.

When the program finishes running, `main` returns with a return code of 0 if there were no errors and with a return code of 1 otherwise.

Do *not* print any extra newline characters or any prompts for the user.

How do you end standard input?

One way to run your program is to redirect the input to come from a file. Using that approach, it is clear when the file ends. But how do you close standard input when it is coming from the keyboard? You manually indicate the end of standard input from a keyboard by pressing Ctrl-D (on Unix) or Ctrl-Z (on Windows) and enter.

Submission

Use git to submit your final `compile_all.sh`, `my_commands.sh`, `invest.c`, `score_card.c`, `phone.c` and `phone_loop.c` -- 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. Do NOT add or commit executables to your repository. We will build executables by compiling your code as part of testing it.

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`.