COMP211 - Lab 00 - A Tutorial Introduction

Abstract

This lab is intended to give you practice working in a command-line interface shell. In the process of completing this lab you will begin gaining comfort using `vim` to edit `C` code and using the `gcc` compiler to compile your programs into executable binaries. Additionally, you will use `git` to form commits, push your changes to GitHub, and submit for autograding on Gradescope.

The concepts in this lab extend on concepts covered in Chapter 1 of *The C Programming Language*. Before starting this lab you should complete the reading, work through its examples, and complete the GRQs.

Setup

Starter Repository

Accept the following GitHub classroom assignment: https://classroom.github.com/a/E9IU-Fsj

With your project repository open on GitHub, click the green "Clone or Download" button. Choose the HTTPS option, for now. Copy the URL to your repository to your clipboard that begins with https.

In a `comp211` shell session, which you can begin by starting Terminal (MacOS) or Windows Terminal/PowerShell (Windows) and then:

```
cd learncli211
# Mac:
./learncli.sh
# Windows:
./learncli.ps1
```

Once your container's session begins, you can clone your project's repository with the command `git clone <URL>`. Replace `<URL>` with your copied repository URL. (Right click to paste in Windows PowerShell.) Change your working directory to be the cloned directory: `cd lab00`. Use `pwd` to confirm you are working in the project repository.

Part 0. Hello, world

In your repository, make a new directory named exactly `0-hello-world`. Change your working directory to be this subdirectory. For reference on how to carry out either of these tasks, please refer to *Chapter 1: The Sorcerer's Shell*.

Next, you'll want to edit a new file named `hello.c`. To begin editing this file in `vim`, simply run the command:

```
vim hello.c
```

Each source file in COMP211 will begin with the standard header comment below. Note this header is checked by the autograder for an exact match. Please be sure to format your PID as a single 9-digit number with no spaces nor dashes. Additionally, be sure your capitalization and punctuation of the honor code pledge are correct. Since we do grade manually for style we do not include names on code listings to avoid biasing the grading.

```
// PID: 9DigitPidNoSpacesOrDashes
// I pledge the COMP211 honor code.
```

`hello.c` Requirements

The purpose of `hello.c` is to slightly extend the book's implementation of the same program on Page 6. Your implementation should print "hello, world" on one line, "welcome to c!" on another line, *and* return `EXIT_SUCCESS`. To return `EXIT_SUCCESS` you will need to import `stdlib.h`, the header file which defines this constant. When `main` returns `EXIT_SUCCESS` it indicates the program completed successfully via a success *exit status*. We will explore the idea of *exit statuses* later this semester. Additionally, in 211 we expect all function return types to be defined and `main` should return an `int`.

Compiling and Executing

You can compile and execute your program with the following shell commands:

```
gcc -Wall -std=c11 hello.c
./a.out
```

Once your program compiles without warnings and meets the requirements, you can continue on.

Make a Commit

Form a commit that includes the `hello.c` source code file with a meaningful commit message. You should not add `a.out` as it tends to be bad practice to include compiled files in source code repositories. Push this commit to your GitHub repository. Open your GitHub repository in a web browser and confirm you see the `0-hello-world/hello.c` file in your repository.

Submitting the Project

You will submit to autograding on Gradescope via a GitHub integration. You must make a commit and push your commit to GitHub in order to resubmit on Gradescope.

Navigate to the project on Gradescope. The first time you attempt to upload your work from GitHub to Gradescope you will need to authorize Gradescope to have access to your GitHub repositories. Once you've done that, you can create a submission by selecting your private repository from the Repository selector and `master` from the branch selector. Once uploaded, the autograder should run with feedback shortly.

You should earn full autograding credit on the `0 - hello.c` tests before continuing to Part 1.

Part 1 - Stripping `bash` Comments

In the next part of this lab, you will write a program that reads from standard input and strips out `bash`-style comments. In the `bash` scripting language, comments begin with a `#` symbol and continue to the end of a line.

To begin, change your working directory to be the back at Lab 00's base, as in the parent directory of `0-hello-world`. Then, change your directory to be `1-bash-comments`. If you list the files in this directory with `ls` you will see there are some sample script files already established for you for testing purposes which will be explained shortly.

You should begin your program work in a file named `strip-comments.c` in the `1-bash-comments` directory mentioned above. This program will be similar to others found in Chapter 1 of *The C Programming Language* that process input using `getchar` character-by-character and maintain information about states (such as in a *comment* state or not).

Your program should read input character-by-character. It should print out all characters verbatim until a comment is encountered and suppress further output until the end of a line is reached. If the comment character, a hash symbol `#`, is found inside of a double quoted string literal, then the comment character should be ignored.

After compiling your program, you can use the `cat` program to pipe the test script files into `a.out` for testing purposes. These test scripts are written in order of difficulty. You are encouraged to get your program to match the first file's expected results before moving to the next.

Expected output processing `00-verbatim.sh`. The first line of output simply demonstrates the original content of the script files:

```
learncli$ cat 00-verbatim.sh
echo demo script has no comments
echo nor string literals
learncli$ cat 00-verbatim.sh | ./a.out
echo demo script has no comments
echo nor string literals
```

Expected output processing `01-comments.sh`:

```
learncli$ cat 01-comments.sh
echo this is code # this is a comment
# this entire line is commented out
echo another command # another comment
learncli$ cat 01-comments.sh | ./a.out
echo this is code

echo another command
```

Expected output processing `02-strings.sh`:

```
learncli$ cat 02-strings.sh
echo "this is a string"
echo "so is # this one"
echo "##s in stringss are ok" # but this is commented out
learncli$ cat 02-strings.sh | ./a.out
echo "this is a string"
echo "so is # this one"
echo "##s in stringss are ok"
```

Once you are ready to submit to autograding, add `strip-comments.c` to stage and make a commit. Push to GitHub and resubmit to Gradescope.

Part 2 - Character Counting

For the final part of this lab, you will write a program that keeps counts of alphabetical characters from standard input and, after `EOF` is detected, displays a summary of the number of occurrences, percentage, and a vertically-oriented histogram of relative frequencies.

Here is an example of a simple input and your expected output:

```
learncli$ echo "Cackalack!" | ./a.out
Counts:
a: 3
c: 3
k: 2
l: 1

Percents:
a: 33.3%
c: 33.3%
k: 22.2%
l: 11.1%

Chart:
| |
| |
| |
| |
| |       |
| |       |
| |       |
| |       ||
| |       ||
| |       ||
abcdefghijklmnopqrstuvwxyz
```

The formatting of each area is expected to match exactly.

To begin, change your working directory to be the back at Lab 00's base, as in the parent directory of `0-hello-world`. Then, make a new directory named `2-char-count`, and in that directory create a file named `char-count.c`

2.1 - Counts

Your program should count occurrences of alphabetical, English letters `a` through `z`, case insensitive. It should ignore all other characters except for the `EOF` character which, when encountered, should display the counts of *only* the alphabetical characters with at least one occurrence. The expected format of this output is, for each character and count on its own line: `<char>: <count>`

2.2 - Percents

After outputing character counts, your program should output the each character's percentage of the alphabetical characters encountered. For example, in the example input "Cackalack!" there were were 9 alphabetical characters and 3 of them were the letter `c`, so `c`'s percentage was `33.3%`. The expected format of this output is, for each character and percentage on its own line, each percentage should have *one digit of significance* and be followed by the `%` symbol: `<char>: <percent>%`

Refer to the book for reference on `printf`'s formatting of a floating point value's decimal digits.

2.3 - Frequency Bar Chart

The final challenge of this lab is to produce a *vertical* bar chart of relative frequencies, as shown in the prior example output. It should be *scaled* such that the tallest bar in your chart is *always* 10 lines tall. Use the vertical bar character `|` when drawing the bars. When the height of a bar has a decimal component, you should always *truncate* it (always round down). Unlike the previous two parts, there should be an entry for every alphabetical character to form the x-axis of the chart as shown in the previous example, even if it has no bar.

Other Examples

Once your implementation is complete, try using `/usr/share/dict/words` as an input, or the following command which counts the characters of Shakespeare's complete works by downloading them from a web resource via the `curl` program:

```
curl -s http://www.gutenberg.org/files/100/100-0.txt | ./a.out
```

You should find 481,437 `e` characters in the file above!

Helpful Hints

Each part of this lab is an extension of or readaptation of concepts introduced in Chapter 1 of the book. If you are having trouble knowing where to start, read back through Chapter 1 and start with some psuedo-code.

For the frequency bar chart, refer to the book's example of `truncation` with integer division and casting from `double` to an integral type.

Grading (25 points)

(20 Points) Autograded Functionality: feedback on this is provided as soon as Gradescope grading completes. You are encouraged to continue improving your code resubmiting until achieving full credit.

(5 points) Style: manually graded by course staff, after the late deadline, on the following criteria:

- Use of curly braces around *all* control statement bodies (`if`, `while`, `for`, etc)
- All magic numbers are defined as constants (0, 1, and 100.0 are acceptable number literals to use in code)
- Part 2 must keep character counts using arrays, not individual variables per letter *and* be organized into simpler functions, not entirely written in the `main` function
- Meaningful names of variables and functions are chosen
- No global variables

Honor Code and Collaboration

No collaboration is permitted on this assignment beyond the undergraduate teaching assistants working for the course. No external resources should be used beyond the course text book.