Shell and Debugging Assignment (Evaluation)

CSCI 221: Computer Science Fundamentals 2

Fall 2024

This assignment is an opportunity to test your understanding of shells and debugging. Feel free to collaborate with others and use resources, but all code and writeups must be your own. For each of the questions, you should also write code to test your results and a makefile that compiles your code. You can use one makefile for all compilations, or you can have separate makefiles for different questions. To submit your assignment, please submit your code, makefile(s), and a pdf file containing your answers.

Due Date: Friday, September 19th at 1:00 pm.

1. (12 points) **Structure and Makefiles.**
   (a) (8 points) Modify your code from the Intro C evaluation assignment to follow the style and structure that we discussed in class. In particular, no file should have code for more than one question. Furthermore, you should be able to create an executable file for each individual question. Body and header files should have names that exactly match the name of the function specified in that question. Body files used to generate executables should have the function name followed by "_testing".

   (b) (4 points) Create an additional code file that calls the function from each question (the inputs and how you use them don't matter). Add a rule to your makefile that generates an executable file from that code.

2. (15 points) **Shell.**
   (a) (3 points) What flags are being passed to the following command? What are the arguments associated with each flag?
   `command --abc def -gh ij`

   Solution: In the given command, (1) `--abc`, (2) `-g`, (3) `-h` are the flags. `def` is the argument associated to the flag (1), and `ij` is the argument being passed to (2) and (3).

   (b) (4 points) Assume that you just installed (or wrote) a new library, and you would like to be able to run an executable from that library using your shell with one command that did not depend on your current working directory. How would you accomplish this?

   Solution: Say the executable is called `exec.out` and is located at ∼`/code/CS221/exec/`. The way to run the executable from anywhere is to (1) add the directory to the executable to `$Path`:

```
export PATH = "/code/CS221/exec/":$PATH
```

   This will add the directory of the executable to the `$PATH` environment variable. So whenever we run the executable, the compiler will check for the executable in the directory.

   (c) (4 points) Shells have many configurable settings that allow users to modify how they interact with the shell. For bash, these settings are stored in the `.bashrc` file in your home directory. Other shells will usually have a similar file. Find and open this file. What are some settings that are important to how you interact with the shell? Customize your shell to be useful to you. Submit your modified settings file.

   Solution: I added an additional function in the end:

```
# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/Users/sajidmahamud/anaconda3/bin/conda' 'shell.zsh' 'hook' 2>
    /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/Users/sajidmahamud/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/Users/sajidmahamud/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/Users/sajidmahamud/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<

# define a function called run to make life easy

run(){
        File=$1
        gcc $File.c -o $File.out
        ./$File.out
}
```

(d) (4 points) How can you count the number of instances of the word "that" in the first 50 lines of the file `Logistics.tex` using a single command? (The file `Logistics.tex` is the tex file from the very first assignment. It's on Moodle if you want to test your code.)

Solution: Assuming the working directory is the directory where `Logistics.tex` is stored, we'll use the following command
`head -n 50 Logistics.tex | grep -i ''that'' | wc -l`

3. (18 points) **Version Control.**

(a) (2 points) What is the difference between creating a repository and cloning a repository?

Solution: The main difference is in the purpose: you create a repository for an entirely new project, whereas you clone a repository to work (or continue to work) on an existing project.

(b) (2 points) What is the difference between a staged file and a committed file?

Solution: Both relate to the status of new changes made to a file. Staged files are at a preparatory state before they are committed. Once a staged file is committed, a snapshot of the changes in the file is stored within the repository. The other difference is about the temporality of the changes. Say you've made changes to `file.txt` and you've already staged it. Before committing to store the new changes into repository history, you realize that something crucial was missing from the file. Now, you can go ahead and `restore --staged` the changes from the preparatory state, without affecting the original file in the working directory.

(c) (2 points) If you realized that a previous implementation in your project was better for what you were trying to do now, and you wanted to return the entire repository to the state at that time, what command should you run to do this?

Solution: I will ignore the ambiguity of the phrase 'return the entire repository' to the state at that time and interpret it as wanting to undo all changes made to the repository permanently. The way to do it is `reset --hard [commit ID]`. This is assuming I had a `commit` made for the time in question.

(d) (2 points) If you made a mistake while working on a project and wanted to undo the changes you made to a particular file, what command should you run to do this?

Solution: Again, I will ignore the ambiguity of the phrase "undo changes you made" because it does not mention whether the changed file was staged or committed or pushed to remote. Let's call the file: `file`. If the changes weren't staged then `git restore file` will remove the changes. If the file was staged, `git restore --staged file` will take the file out of staging area and doing `git restore file` will revert the changes. If the `file` was committed and I want to undo the changes back to the previous commit then we can use: `git reset --hard HEAD 1`.

(e) (4 points) Assume that you are working on a project using multiple different machines which each have a copy of the project. If you make changes on one machine, then make changes on another machine prior to synchronizing your repository, you will have to merge your changes. Explain the process of merging changes, including what happens if your different versions changed the same file.

Solution: One way to resolve the issue is to first `commit` and `push` changes made from `device 1` to the remote repo. Then on `device 2`, go ahead and `commit` the changes, but don't `push` changes to the remote repo yet. Do `git pull origin main` (assuming that the changes are all in one branch). Then there are two possibilities: (1) changes were made to different files and `merge` happens automatically, (2) changes were made to the same file and `merge` conflict needs to be resolved manually by going to the file. Once that's done, we can `add`, `commit` and `git push origin main` the changes to the remote repository.

(f) (3 points) What is a pull request?

Solution: The pull request is usually followed by an independent `git push` from a `branch` other than `main`. Say you are working on a branch called `branch grader` and you made changes to the repository on the branch `sajid`. After making the changes, you `push` the changes to the remote repo on `github`. You can then request the changes to be added to the `main` branch (or any branch you select as the target). This is what a pull request is.

(g) (3 points) What Git command should you use to compare two different commits in the same repository? How does it work?

Solution: Say I want to compare changes between `commit1` and `commit2`. The command to use is `git diff commit1 commit2`. Doing so reveals the list of comparisons between the commit in terms of (1) new additions: which lines are added, (2) deletions: which lines are removed, (3) modifications: which lines are modified.