Assignment 0
git'n Started

Prof. Darrell Long
CSE 13S – Winter 2020

Due: January 12th at 11:59 pm

1 Introduction

It's nice if people can finally loosen up a little bit and just go out and laugh at silliness. I mean, people take themselves way too seriously sometimes. git*'er done!*

—Larry the Cable Guy

The aim of this first assignment will be for you to set up your git repositories and gain an understanding of how git works. We will review several git commands that you will help you in the long run. This document will be helpful for troubleshooting git issues in the future and also includes the submission policy. You will find this quite helpful in the future if you ever have any issues with git or submitting. *Ideally,* this assignment is to be completed during your discussion section.

2 Your Task

The story of life is quicker than the wink of an eye, the story of love is hello and goodbye... until we meet again.

—Jimi Hendrix

We will be creating a simple **C** program and compiling it. The program will simply print ``Hello World!''. This tutorial can also be found in Chapter 1 §1.1 in your textbook, *The C Programming Language* by Kernighan & Ritchie. At this point, you will want to be in your asgn0 directory as this will be your *current working directory*.

1. Connect to unix.ucsc.edu using an ssh client.
2. Create the program source hello.c with your text editor of choice. Make sure you do this with a text editor like vi or emacs. Notepad is *not* a text editor. For example, you would type the following into your terminal:

```
vi hello.c
```

3. Include the headers for the stdio.h library. This is needed to use printf.

```c
#include <stdio.h>
```

4. Set up your `main` function.

```c
int main(void) {
  return 0;
}
```

5. Inside the `main` function (in between the curly braces) will be where you will put `printf`. It is *crucial* that your output matches the one given here. You will be docked points otherwise.

```c
printf("Hello World!\n");
```

6. To compile and run this program, exit your text editor to return to the command line. For example, if you are using `vi`, you would hit the `esc` key, type "`:wq`" and press the Enter key. This saves your work and exits the text editor.

7. You should be back on the command line. To compile your program, type the following line into the command line:

```
clang -Wall -Wpedantic -Wextra -Werror hello.c -o hello
```

   This is also known as the "take no prisoners" compiler flags. Simply put, they catch pretty much everything that a compiler can catch (there are a few more esoteric warnings that can be enabled). For example, the "`-Werror`" flag turns all warnings into errors. The "`-Wextra`" option warns about expressions that have no side effects and whose value is discarded.

8. If you've done everything correctly up to this point, the compilation process should run silently and return no errors. However, if you do run into any errors, Google, Piazza and discussion sections will be your best friends.

   The most likely problem is that you did not use a *text editor* on `unix.ucsc.edu`—contrary to our explicit advice.

9. After successfully compiling your program, there is now an executable file called "`hello`". To run your program, simply type the command "`./hello`" and it should print "`Hello World!`" onto the console. Our executable is called "`hello`" and "`./`" is added to the front of it to indicate the executable is in our current working directory.

10. Congratulations! You have now written your first program.

Now, your next step is to submit your program along with the `CHEATING.pdf` through `git`. The `CHEATING.pdf` can be found under files on Canvas and/or under Piazza resources. Note: You do not create a PDF file by simply appending `.pdf` to its name.

Academic honesty is very important in computer science, and life in general. The goal of this course is for you to learn the material, not simply for you to get a mark on your transcript saying you passed the class. All students in the class must sign and turn in an acknowledgment that they understand the cheating policy for the class. We will not accept or grade any assignments from a student unless he or she has turned in the `CHEATING.pdf`. We encourage you to ask for clarifications in the academic policy if you have any questions.

3 Version Control

Dire Straits is a great band. Someone tells you they like 'Brothers in Arms' and immediately you know they're a stupid annoying git.

—Alexei Sayle

git allows you to maintain multiple versions of your source files, also known as version control. Version control lets you maintain a source code tree that in the case of an accident, you can obtain your previous code back. If your computer suddenly decided to explode while you were working on your code, you can pull the latest committed version of your code. It allows you to put "checkpoints" into your code development.

It provides a set of commands that you can use to maintain a version controlled repository. A repository is a directory that is either on your local computer or on a server. A repository is also where you store your files. With certain git commands, you can maintain and manipulate the files in a repository. For this class, you will be using git to turn in all assignments.

3.1 Getting Started

To get set up, you will first need to clone the git repository on your local computer. Open up your terminal and type:

```
git clone https://gitlab.soe.ucsc.edu/gitlab/cse13s/winter20
```

This command will ask you for your permission to authenticate with the server. Afterwards, it will clone your repository onto your machine into a directory named `cse-13s/` in the current working directory. Now, we will review some useful git commands that will be essential in this class. Additional information on git can be found at this link:

`https://gitlab.soe.ucsc.edu/gitlab/help/gitlab-basics/README.md`

3.2 `git clone`

This command clones a repository from a server onto your local machine. This downloads a copy of the repository which is stored on a server for local editing. Meaning, any changes that need to be sent back to the server will need to be *added*, *committed* and *pushed*. Here is an example of cloning over ssh:

```
git clone user@somemachine:path/to/repo
```

3.3 `git init`

This command initializes the directory to become a repository and allows you to add files. When cloning a repository, this command does not need to be used since the repository has already been initialized.

3.4 `git add`

This command allows you to add files into your repository and stages them to the git source tree. Any file that has been changed since the time it was last added needs to be added again.

```
git add file1 file2
```

Keep in mind, adding files with this command does *not* commit them. You still need to commit the changes with the `git commit` command.

3.5 `git commit`

This command creates a checkpoint for each file which was added using the previous command, `git add`. You can think of it like capturing a snapshot of the current staged changes. These snapshots are then safely committed. Each commit has an unique commit ID along with a message about the commit.

```
git commit -m "A short informative message about any changes"
```

To commit all the changed files, you can use the command `git commit -a` which can also be combined with the `-m` option. This will only commit files that have been added and committed at least once before. Without the `-m` flag, you will be prompted into an editor to enter your commit message. A forewarning: don't commit rude comments – the TA's will see them.

You should commit working versions of your code frequently so in the case you mess something up, like accidentally deleting your code, you can use `git checkout HEAD` to revert to the most recent commit.

3.6 `git checkout`

This command allows you to navigate between branches created by `git` branch. It can help you undo changes in the case you mess up and come to the rescue. Checking out a branch is similar to checking out old commits. The files in the current working directory is updated to match the selected branch or commit ID. It also tells `git` to store all new commits on that branch.

```
git checkout <branch>
```

3.7 `git push`

This command pushes all of your local commits to the upstream repository. It pushes all of your changes to the directory which is stored on-line. You *must* do this to turn in your work for this class. If you do not run this command after committing, *none* of your work will be turned in.

3.8 `git pull`

This command fetches and downloads content from a remote repository. Your local repository is immediately updated to match the fetched content. `git pull` is actually a combination of `git fetch` followed by `git merge`. The first half of `git pull` will execute `git fetch` on the local branch that HEAD is pointed at. After the contents are fetched, the second half of `git pull` will merge the work-flow creating a new merge commit ID and HEAD is updated to point to the new commit.

3.9 `git ls-files`

This command lists all files in the current directory that have been checked into the repository. This will be useful for making sure you have submitted all required deliverables for each assignment.

3.10 `git status`

This command provides a status of which files have been added and staged for the next commit, as well as unpushed changes.

3.11 `git log`

This command provides a list of the commits that have been made on the repository. It provides access to look up commit times, messages, and IDs.

4 Deliverables

If there was no Black Sabbath, I could still possibly be a morning newspaper delivery boy. No fun.

—Lars Ulrich

For this class, you will be turning in all of your work through `git`. All the files you need to turn in for an assignment will be found and listed in the Deliverables section of every assignment PDF. Files will need to be turned in the corresponding assignment repository, committed, and pushed. To verify that you are doing this correctly, you can check your commit appears correctly at:

`https://gitlab.soe.ucsc.edu/gitlab/cse013s`

You will need to turn in:

1. `asgn0/CHEATING.pdf`
2. `asgn0/hello.c`

5 Submission

The cost of freedom is always high, but Americans have always paid it. And one path we shall never choose, and that is the path of surrender, or submission.

—John F. Kennedy

Now that you have learned about some useful `git` commands, it's time to put them to use. The steps to submitting assignments will not change throughout the course. If you ever forget the steps, refer back to this PDF. Remember: *add, commit,* and *push*! In the case you do mess something up, *don't panic.* Take a step back and think things throughly. The Internet, TAs and tutors are here as resources.

1. Add it!

   ```
   git add CHEATING.pdf hello.c
   ```

   As mentioned before, you will need to first add the files to your repository using the `git add <filenames>` command. You will be submitting these files into the `asgn0` directory.

2. Commit it!

```
git commit -m "Your commit message here"
```

Changes to these files will be committed to the repository with `git commit`. The command should also include a commit message describing what changes are included in the commit. For your final and last commit for submission, your commit message should be "final submission".

3. Push it!

```
git push
```

The committed changes are then synch'd up with the remote server using the `git push` command. You must be sure to push your changes to the remote server or else they will not be received by the graders.

Your assignment is turned in *only* after you have pushed. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

6 Supplemental Readings

The more that you read, the more things you will know. The more that you learn, the more places you'll go.

—Dr. Seuss

- *Version Control with Git* by Loeliger & McCullough ← Read this! Now!
  - Chapter 3 – Getting Started (pg. 22–25)
- *The C Programming Language* by Kernighan & Ritchie ← It is a *huge* mistake to not read this!
  - Chapter 1 §1.1
- *vi and Vim Editors* by Robbins & Lamb
  - Chapter 1 §1.4 & §1.5