Assignment 0
`git`'n Started

Prof. Darrell Long and Dr. Kerry Veenstra
CSE 13S-01 – Winter 2023

Due: January 15th at 11:59 pm

1 Introduction

The aim of this first assignment will be for you to set up your GitLab 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 should be completed during your discussion section, assuming that you already have installed Ubuntu in a VM.

2 GitLab

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

The deliverables for each of your assignments will be maintained through your GitLab repository. GitLab is a service coupled with the version-control capabilities of `git`. `git` allows you to maintain multiple versions of your source files, also known as version control. Version control is the practice of tracking and making changes to code, such that in the event of some accident while coding, it is always possible to restore your code to a previous state. `git` is used through a set of commands within a repository, a version-controlled directory that stores your files.

2.1 Setting Up SSH Keys

You will first need to *clone* your GitLab repository. It is highly recommended that you use `git` over SSH rather than HTTP. SSH causes GitLab to use secure communication between `git` and its servers using SSH keys, and that using the SSH protocol allows for authentication *without* the need to enter your username or password each time.

SSH keys come in *pairs*: a *private* key and a *public* key. Data encrypted with some public key can only be decrypted with the corresponding private key and vice versa; public key cryptography. (All commands below are run in an Ubuntu Terminal. The $ is a prompt; type what follows.) To generate an SSH key pair

```
$ ssh-keygen
```

The SSH keypair generated by the default prompt answers will be sufficient for your needs for use with GitLab. (Make sure you add the *public* key of the generated key pair to GitLab and that it is an RSA key.)

To print your public key so that you can copy it to your clipboard, enter the following:

```
$ cat ~/.ssh/id_rsa.pub
```

Next follow steps 2 through 9 of this link, but see below for what to do at step 6.

https://git.ucsc.edu/help/user/ssh.md#add-an-ssh-key-to-your-gitlab-account

For step 6 (pasting the public key), you'll first copy the block of text that you cat'ed above. That is, after selecting the public key in the Ubuntu Terminal, copy it by right clicking and then selecting Copy.

After adding the key, you will be ready to clone your GitLab repository. For more in-depth instructions on generating and adding SSH keys, as well as other GitLab basics, please refer to this link:

https://git.ucsc.edu/help

2.2 Cloning Your Repository

To clone your repository, run the following command, substituting `<CruzID>` with your CruzID:

```
$ git clone git@git.ucsc.edu:cse13s/winter2023-section01/<CruzID>.git cse13s
```

You will be prompted for permission to authenticate with the server. When permitted, the command will clone your repository onto your machine into a directory named `cse13s` in the current working directory. Use the `cd` command to enter the `asgn0` directory in your cloned `cse13s` repository to start your work for assignment 0.

```
$ cd cse13s/asgn0
```

3 Hello World!

You will be creating a simple **C** program which will simply print "`Hello World!`" You can find also find a tutorial of this program in Chapter 1 §1.1 in your textbook, *The **C** Programming Language* by Kernighan & Ritchie.

1. Make sure you are in the correct directory: `asgn0`. You can check your *current working directory* using this command:

   ```
   $ pwd
   ```

2. Create the program source `hello.c` with your text editor of choice. This means text editors such as `vi` and `emacs`. Notepad and Word are *not* text editors.

   A good introduction to `vi` is in Chapter 5 of *Learn Enough Developer Tools to be Dangerous*. You can read this book online for free by searching for it on the UCSC library's home page.

   To open up `hello.c` for editing with `vi`:

   ```
   $ vi hello.c
   ```

3. Include the header for the `<stdio.h>` library. This is needed by the `printf()` function that prints formatted strings to `stdout`, what you think of as the console. (In the black code listing blocks below, the numbers on the left are not part of the C program. They are just line numbers that make it easier to discuss specific lines of the program, so don't enter them into your text file.)

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

4. Type your `main()` function. Every **C** program *must* have a `main()` function which returns an `int`. A return of 0 indicates program success, and a non-zero return indicates the occurrence of some error.

hello.c
```c
1 #include <stdio.h>
2
3 int main(void) {
4     return 0;
5 }
```

5. In `main()` (between the curly braces) is where you will type the print statement. It is *crucial* that your print statement matches the one given here. *You will be docked points otherwise.*

hello.c
```c
1 #include <stdio.h>
2
3 int main(void) {
4     printf("Hello World!\n");
5     return 0;
6 }
```

6. Save your work and exit your text editor to return to the command line. With `vi` this means entering normal mode by hitting `esc` and entering the `vi` command "`:wq`" to save and quit.

7. You should now be back on the command line. You should now compile and run your code to verify its correctness. To compile your code, run:

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

This will compile your code with the compiler flags required by the class. `clang` is the **C** compiler that we will be using—not `gcc`, not `cc`. You *must* use `clang`. The `-Wall -Wextra -Werror -Wpedantic` arguments are the set of compiler flags you must use when compiling your code. This specific set of compiler flags is commonly referred to as the "take no prisoners" compiler flags. Simply put, together they catch pretty much everything that a compiler can catch (there are a few more esoteric warnings that can be enabled). Here are some links for you to investigate what each flag does:

https://releases.llvm.org/11.0.0/tools/clang/docs/UsersManual.html

https://releases.llvm.org/11.0.0/tools/clang/docs/DiagnosticsReference.html

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, lab sections, and the dicussion forum will be your best friends. Resist the urge to immediately use Google.

9. After successfully compiling your program, there should now be an executable file named `hello` in the current working directory. To list out all the files in the current working directory use `ls`:

   ```
   $ ls
   ```

   To run the `hello` program, enter:

   ```
   $ ./hello
   ```

10. The `.` (usually called "dot") refers to the *current working directory*. Your shell has a `PATH` environment variable, a colon-delimited list of directories that it looks through when you enter a command. Since your current working directory is most likely not in your `PATH`, you must specify the directory that your program can be found in order to run it. If the output of running your program is correct, you should then submit your working *source code* to `git`. You should submit source code *only*: no executables.

    ```
    $ git add hello.c
    $ git commit -m "Adding finished hello.c"
    $ git push
    ```

    The above three commands will add, commit, and push `hello.c` to `git`. In-depth description of each of these commands will be provided in the following section. To verify that `hello.c` was added, check your repository (remember to type your own CruzID instead of `<CruzID>`):

    `https://git.ucsc.edu/cse13s/winter2023-section01/<CruzID>.git`

    Only in this case do you perform one commit at the end. In general, you should commit after every significant change.

11. The second file to be submitted for assignment 0 is your signed `CHEATING.pdf`. This file can be found on Canvas and/or under the discussion forum resources. You will submit `CHEATING.pdf` the same way you did `hello.c`: adding, committing, then pushing. Pushing to `git` under your SSH key is considered a signature.

4 An Essay

Engineers must communicate well, both orally and in writing. In this part, you will write a 2–3 page essay on "Just Scoring Points," a contribution by professor Walter R. Tschinkel in *The Chronicle of Higher Education*. You will find a copy of Prof. Tschinkel's writing in:

`https://git.ucsc.edu/cse13s/winter2023-section01/resources.git`.

Read "Just Scoring Points" and tell us what you think about it. Do you agree or disagree with Tschinkel's opinion? There is no wrong answer; we want you to read and engage with Tschinkel's work and communicate your ideas. You may use Word, Google Docs, or as a budding Computer Scientist, LaTeX. A simple LaTeXtemplate is provided in the resources. However, regardless of the editor that you use, the file you turn in *must* be a PDF.

Simply adding the suffix `.pdf` does not magically transform a file into PDF; it must be created. All of the word processing programs are rather like compilers for a programming language: they transform text into a printable document.

You are now probably wondering: *This is supposed to be a Computer Science course! Why am I writing an essay?* The answer is simple: Computer Scientists write programs, but they write more English text than they do programs. You need to be able to express yourself in writing clearly, concisely, and convincingly.

5 Git

The following commands are used through `git` for version control. For this assignment, you will have used the `clone`, `add`, and `push` commands. This section will serve as a brief description and use of frequently used `git` commands that you will most likely use throughout the quarter, if not your entire career as a computing professional.

5.1 `git config`

This command lets you set configuration variables that tune `git` to operate the way you want it to. The main things you will likely want to do when you get started with `git` are establishing your identity, as well as the default text editor when typing up longer commits.

Perform the following command to set your name and email address. Notice the `--global` in the command. That is a *command-line option* and indicates that the following configuration should be used globally in every `git` repository you have. Unless otherwise specified, configurations by default are applied only to the local repository.

```
$ git config --global user.name "<your name>"
$ git config --global user.email "<your email>"
```

To set your default editor as `vim`:

```
$ git config --global core.editor vim
```

To check all the configurations simply run:

```
$ git config --list
```

To check the value of a specific key, or setting, just supply it as the sole argument after `git config`. For instance, to check the configured email:

```
$ git config user.email
```

5.2 `git help`

When starting out with `git`, you may find yourself frequently needing to refresh your memory on certain commands. The command `git help` will prove invaluable in this regard. There are three ways to display the `man` page for any `git` command:

```
$ git help <command>
$ git <command> --help
$ man git-<command>
```

For example, to view the `man` page for `git clone`, the subject of the next section, any of the following can be run:

```
$ git help clone
$ git clone --help
$ man git-clone
```

A `man` page (short for manual page) is software documentation for tools and programs found on UNIX systems. To view a `man` page:

```
$ man <function, program, tool>
```

These manual pages are typically divided into sections, depending on their respective purposes. General commands are found in section 1, system calls in section 2, and library functions, such as the `printf()` function used in this assignment, are found in section 3. So, to view the `man` page for `printf()`:

```
$ man 3 printf
```

5.3 `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
```

5.4 `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.

5.5 `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.

5.6 `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. You can display the history of commits you can check out to with `git log`.

```
$ git checkout <branch>
```

5.7 `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.

```
$ git log
```

5.8 `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.

```
$ git push
```

5.9 `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.

```
$ git pull
```

5.10 `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.

```
$ git ls-files
```

5.11 `git status`

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

```
$ git status
```

6 Honesty

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 they have turned in the `CHEATING.pdf`. We encourage you to ask for clarifications in the academic policy if you have any questions.

7 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 added to the corresponding assignment directory, committed, and pushed.

You will need to turn in:

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

8 Submission

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 essay.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 sync’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*.

4. Submit the commit ID to Canvas. Your committed and pushed changes will have a unique hexadecimal string as a commit ID. You can see this with the command:

   ```
   $ git log
   ```

   This command will show you an output like this with the commit ID after ‘commit’.

   ```
   commit 0a2d3b663a962cf86af5ad82c80baf1cd71f7766 (HEAD -> main, origin/main, origin/HEAD)
   Author: Changyu Bi <changyubi@meta.com>
   Date:   Thu Jan 5 12:10:02 2023 -0800

   Fix some unit test failure in ExternalSSTFileBasicTest (#11070)
   ```

   Copy the string: For example, `0a2d3b663a962cf86af5ad82c80baf1cd71f7766` in *this* case, and paste it into the text box on Canvas for assignment 0, and submit.

9 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
- *Learn Enough Developer Tools to be Dangerous* by Michael Hartl
  - Chapter 5 (vi)
  - Chapter 8 (git)

Honni soit le singe qui rit.