Assignment 0
git'n Started

by Prof. Darrell D. E. Long
edits by Dr. Kerry Veenstra
for CSE 13S – Spring 2024

Due: Tuesday, April 9th at 11:30 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.

Below you will be instructed to run various commands. You will be running these commands under Ubuntu Linux in your VM. Don't try to run them under Windows!

2 Set Up Source Code Control

The deliverables for each of your assignments will be maintained through your `git` "repository" (often called a "repo"). We are using GitLab, which 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 You Must Create a GitLab Account

You must create an account at `git.ucsc.edu`. Your GitLab account must use the same email address that you use with Canvas. Once you create your account, your repository for the class will be created automatically during normal waking hours. (That is, the background task that creates the repositories is running on a TA's laptop, and so that laptop needs to be active for the repositories to be created.)

2.2 Set Up SSH Keys

You will first need to *clone* your GitLab repository, which means getting a copy onto your laptop. It is highly recommended that you use `git` over SSH rather than HTTP. SSH, the *Secure Socket Shell,* is a protocol that provides secure network communication. Using `git` over SSH allows for user authentication using SSH keys, ridding the need to enter your username or password each time you want to push or pull changes to your GitLab repository. Other tools that use SSH include `scp` and `sftp`. The former is a *Secure Copy Protocol* designed to transfer files to and from servers over an SSH connection. The latter is a *SSH File Transfer Protocol* and is typically used as a remote file system protocol.

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. You may find it useful to keep a copy of your SSH keys somewhere — like a USB stick. As long as you have access to the keys, you can securely authenticate with `git`.

Log into Ubuntu Linux on your VM. Then generate an SSH keypair by executing this command. NOTE: the `$` at the beginning of the line below represents the command prompt that is printed by Ubuntu Linux, and so, for example, you don't type `$ ssh-keygen`, you type the part that follows: `ssh-keygen`.

```
$ ssh-keygen
```

This assumes the use of Ubuntu Linux, UNIX, or macOS — attend section if you wish to learn how to generate a key pair on Windows.

The SSH keypair generated by the default prompt answers will be sufficient for your needs for use with GitLab. These keys last a long time, so try not to lose them. 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
```

This specific command uses the UNIX utility `cat`, which is designed to concatenate and print files. The argument supplied to `cat` is the path to your public key. All keys generated by `ssh-keygen` reside under the `ssh` directory. 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/`

2.3 Clone the Resources Repository and Your Own Repository

To clone the resources repository and your repository, run the following commands. (Remember that you don't type the prompts `$`.)

In the last command below, instead of typing `yourcruzid` type the cruzid of your UCSC email address. For example, I typed `veenstra`. You'll be typing something else.

```
$ cd
$ mkdir s24
$ cd s24
$ mkdir 13s
$ cd 13s
$ git clone git@git.ucsc.edu:cse13s/spring-2024-section-01/resources.git
$ git clone git@git.ucsc.edu:cse13s/spring-2024-section-01/yourcruzid.git
```

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 repository to start your work for assignment 0.

```
$ cd yourcruzid/asgn0
```

3 Write 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. Install required software.

   For this class, there are a few tools that you will need in your virtual machine. To learn about what software you need to install and how to install it, see For further references see The Pages under "Using Ubunutu" on Jessie's Gudie (https://13s-docs.jessie.id/usage/). When you submit this assignment, you are expected to have read all pages of this chapter in the guide. If you are ever unclear, please come to section to get help.

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

   ```
   $ pwd
   ```

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

   BTW, I prefer `vim`. I can get work done much more quickly with `vim`, but it takes more training to use it than to use `nano`.

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

   ```
   $ nano hello.c
   ```

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

   ```
   $ vim hello.c
   ```

4. 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.

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

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

```
#include <stdio.h>

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

6. 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. Your program gets graded automatically. A single-character difference in the printed output will cause you to lose points. Is there a comma? Is there a backslash followed by letter `n` like this: "`\n`"? (Not this: "`/n`".)

```
#include <stdio.h>

int main(void) {
    printf("hello, world\n");
    return 0;
}
```

7. Save your work and exit your text editor to return to the command line. With `vim` this means entering normal mode by hitting `esc` and entering the command (indicated with a prefixed colon) "`:wq`" to save ("write") and quit.

8. 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 -Wstrict-prototypes -Werror -pedantic -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 -Wstrict-prototypes -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/14.0.0/tools/clang/docs/UsersManual.html`
`https://releases.llvm.org/14.0.0/tools/clang/docs/DiagnosticsReference.html`

The provided links are for version 14 of `clang`, but it is fine if you have version 15.

9. 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 Slack will be your best friends. Resist the urge to immediately use Google.

10. 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
```

11. To run the `hello` program, enter:

    ```
    $ ./hello
    ```

    What's going on here? The `.` (usually called "dot") refers to the *current working directory.* Although the command shell can run various *commands* (such as `ls`), to run *a program of your own that is in the current directory*, you need to prefix the program with `./`

12. 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.

    To submit your `hello.c` file to `git.ucsc.edu`, perform these steps in your VM:

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

    Some people think that they can save time by using the command "`git add .`" to add all of their files to their repository. Don't do this! You'll end up adding way too many files, most of which never should be under revision control. Only use "`git add` *namedfile*".

    The above three commands will add, commit, and push `hello.c` to `git.ucsc.udu`. 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, substituting your own CruzID for `yourcruzid`:

    ```
    https://git.ucsc.edu/cse13s/spring-2024-section-01/yourcruzid
    ```

    Then click on **asgn0**. You should see the file that you had added to your Git repository.

13. The only other file to be submitted for assignment 0 is `design.pdf`. You create this PDF file any way like, but using a Google Doc or Overleaf both work well. Note: You do not create a PDF file by simply appending `.pdf` to its name. You will submit `design.pdf` the same way you did `hello.c`: adding, committing, then pushing. However, you need to get the PDF file into your VM's storage. You will use a command similar to how you logged into your VM, but called `scp` instead of `ssh`. Attend discussion section for more on this.

14. Turning in an assignment takes four steps. The first three already have been performed (add/commit/push). The fourth step is to get a copy of the Commit ID and submit it to the Canvas assignment for asgn0.

4 Deliverables

As just demonstrated, 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 two files:

1. `hello.c`
2. `design.pdf`

5 Submission

The steps to submitting assignments will not change throughout the course.

1. `git add hello.c design.pdf`
2. `git commit -m "final version"`
3. `git push`
4. `git log -1`

   You'll see a CommitID that looks like: `f66befc567036ef6dd35a498e9b64b42b77174fa` Or instead of using `git log -1` you can use a web browser to visit `git.ucsc.edu` and get the Commit ID there.
5. Submit **your** CommitID (not the example above) to Canvas.

If you ever forget the steps, refer back to this PDF. Remember: *add, commit, push*, and *submit*! In the case you do mess something up, *don't panic*. Take a step back and think about things thoroughly. 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.

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.

4. **Get your CommitID**

   Either visit your repo at `git.ucsc.edu` or use this VM command:

   ```
   $ git log -1
   ```

   This command shows the CommitID of your most recent `git commit`.

5. **Submit the commit ID on Canvas!** As I point out above, you can find the most recent commit ID by using `git log -1` or searching for it through the GitLab web interface. Your assignment is turned in *only* after you have pushed and submitted the commit ID you want graded on Canvas. "I forgot to push" and "I forgot to submit my commit ID" are not valid excuses. It is *highly* recommended to commit and push your changes *often*.

5.1 Backing Up Your Files

The commands above also serve to save back-up copies of your files. In particular, Steps 1, 2, and 3 (`add`, `commit`, `push`) ensure that the git server has a copy of the files that you added. I call this taking a "snapshot" of my work. If something goes wrong, there are commands that will let me get an older version of a (pushed) file after making a mess of things.

Usually I take a snapshot when my program is incomplete but functional, but I also will take a snapshot if I've been working on files for a while, or if I'm about to do some major editing, and I want a form of "insurance" that will let me go back.

5.2 You Should Submit a CommitID More Than Just Once

Steps 4 and 5 involve getting the CommitID of the most recent commit and submitting it to Canvas so that we know which version of your source code to grade. We will grade the versions of your files that are identified by the CommitID that you submit.

Don't wait until the last moment to submit your CommitID! I recommend following this procedure:

1. Aim to have your program in a working state and documented by 10:00 pm before the final deadline. Perhaps you don't have all of the features implemented, but that's okay.

2. Add/commit/push/log and submit the CommitID.

3. Okay! You've submitted something that we can grade! Now you can continue to work on the program and/or the documentation.

4. If you get the program and/or documentation in a *better* state, then repeat the add/commit/push/log and Canvas submission steps. We'll always grade the CommitID that you submit last.

That last point is important. We always will grade the last CommitID that you submit. Don't try to write us an email telling us to grade a prior CommitID. It's gone. You'll need to *re*submit any past CommitID if you want us to grade that one instead of the last one.

Finally, don't write us an email telling us to grade *these* files from *this* CommitID and *those* files from *that* CommitID. We're not going to do that for you. Get all of the files together that you want us to grade and submit them together. See section 8.6 for a command to use that will help you collect files from different CommitIDs for resubmission.

5.3 Checking Your Submission

This is optional, but you can confirm exactly which files we are going to see by creating a copy of your repo and resetting it to the CommitID that you had submitted.

1. Go back to the your `s24` directory. (*Don't* be inside `13s` for the next commands.)

2. You should see the `resources` repo and your repo when you execute `ls`.

   ```
   $ ls
   ```

   You should see `yourcommitid` listed because you are not currently in it.

3. Clone a fresh copy of your repo to a *different name*. (See Section 2.3, but the append `test` to the last part of the command. Instead use, say, `test`.)

   ```
   $ git clone git@git.ucsc.edu:cse13s/spring-2024-section-01/yourcruzid.git test
   ```

4. Go into the test repo.

   ```
   $ cd test
   ```

5. Reset the test repo using the *CommitID* that you submitted for your assignment.

   ```
   $ git reset --hard CommitID
   ```

Now you can look inside the asgn0 directory. The files that you see are the files that we will see.

Is something wrong? Go back into your `yourcommitid` repo (don't stay in the `test` repo) fix any files, and then add/commit/push/log and submit the new CommitID.

Once you've fixed whatever needed fixing, you can check the test repo again using Steps 4 and 5. (You can skip the `git clone` step.

6 Supplemental Readings

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

7 Appendix: The `.ssh` Directory and File Permissions

We'll take a moment here to discuss the contents of the `~/.ssh` directory found on macOS and UNIX-based systems [1], along with basic file permissions on UNIX. There are two files of note, other than generated public and private keys: `~/.ssh/known_hosts` and `~/.ssh/authorized_keys`.

The `~/.ssh/known_hosts` file contains the SSH fingerprints of every remote machine you choose to SSH onto. Whenever you try to SSH onto a new machine, a prompt will appear asking if you want to continue connecting, where an affirmative response will cause the remote machine's SSH fingerprint to be *appended* to the known hosts file.

[1] The `~/.ssh` directory equivalent on Windows is typically `C:\Users\<username>\.ssh`. Your coursework, however, should be done on Ubuntu 20.04 or later.

The `~/.ssh/authorized_keys` file contains the public keys of all remote clients that are authorized to remotely log onto the server using SSH authentication, the server in which is the machine that contains the file of authorized keys.

Each of the aforementioned SSH-related files, including the public and private keys, require specific *file permissions.* File permissions on UNIX are what allow you as a user to read and modify (write) files, as well as what prevents others from being able to read and modify files. File permissions are split into three access groups:

1. User — the owner of the file, typically the user that created the file.
2. Group — a specific set of users.
3. Other — anyone who isn't the user or part of the authorized group.

These three access groups each have three access modes:

1. Read — the ability to view the contents of a file.
2. Write — the ability to modify or delete the contents of a file.
3. Execute — the ability to run a file as a program.

Note that the three access modes were described with files in mind. UNIX directories are also considered files, but the access modes do slightly different things. Read permission for a directory allows permitted users to list files in the directory with a program such as `ls`. Write permission for a directory allows permitted users to modify, create, or delete files in the directory. Lastly, execute permission for a directory allows permitted users to access files within the directory, as well as make the directory their current working directory by using a shell builtin such as `cd` or `pushd`.

The granting and removal of file permissions can be done using the `chmod` utility, typically specifying the desired permissions with three *octal* digits. Why octal? There are exactly two states that each permission can be in: read permission is granted, or it isn't. As a bit, this means either a 0 or 1. Since there are read, write, and execute permissions, this naturally means $2^3 = 8$ possible states, which can be perfectly expressed using one octal digit. Since there are three different access groups (user, group, and other), three octal digits are used. Consider the octal value 6_8. In binary, this is 110_2. The read bit is the most significant bit, the execute bit is the least significant bit, and the write bit is between the two. For this specific example, only the read and write bits are set.

The permissions of your `~/.ssh` directory should be `700`. The leftmost octal digit signifies user permissions, the middle octal digit signifies group permissions, and the rightmost octal digit signifies other permissions. Thus, the permissions on this directory allows only the user to enter it, as well as read and modify files. To explicitly set the permissions as `700` using `chmod`:

```
$ chmod 700 ~/.ssh
```

Read the `man` page for `chmod`, then make sure the permissions for the files in your SSH directory are set as follows:

- `known_hosts` — `644`
- `authorized_keys` — `600`

- any private key — `600`
- any public key — `644`

You can create as many public and private keys with `ssh-keygen` as you want, but you really *only need one.*

8 Appendix: Useful Git Commands

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.

8.1 `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
```

8.2 `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
```

8.3 `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
```

8.4 `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.

8.5 `git commit`

This command creates a checkpoint for each file that was added using the previous command, `git add`. You can think of it as capturing a snapshot of the current staged changes. These snapshots are then safely committed. Each commit has a 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 taken into the default `git` editor to enter your commit message. A forewarning: don't commit rude comments — the TAs and graders will see them.

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

8.6 `git checkout`

This command allows you to set the state of your repository to the state of your repository at the time of a different commit. The reverting of state can be performed per file, meaning that you can use `git checkout` to restore a specific file to its state in a different commit. To checkout a commit:

```
$ git checkout <commit>
```

To checkout restore a file to its state at a different commit:

```
$ git checkout <commit> -- <file>
```

This last command also works to retrieve files that you may have accidentally deleted locally. This alone should provide a good incentive to add, commit, and push changes to your files often.

8.7 `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. Too much information? Then see the next item.

```
$ git log
```

8.8 `git log -1`

This command shows the most recent the commit that has been made on the repository, including the commit time, message, and CommitID.

```
$ git log -1
```

8.9 `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 online. 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
```

8.10 `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 workflow creating a new merge commit ID, and HEAD is updated to point to the new commit.

```
$ git pull
```

8.11 `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.

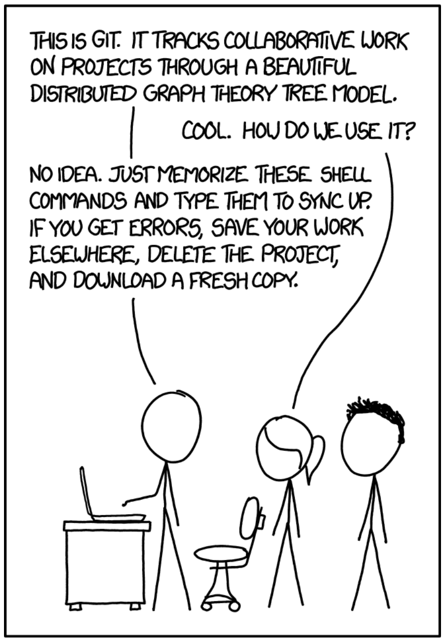
```
$ git ls-files
```

8.12 `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
```

8.13 For when things don't work


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

https://xkcd.com/1597/