Introduction

Deliverable: Make a word doc. You will add screenshots to it. When you finish, save your word doc as a pdf and submit the pdf on Gradescope.

Instructions: Download the zip file from github: using the link on Camino and unzip it into a folder.

- If you have Windows: Launch git bash; this should have come with git.
- If you have Mac: Launch a terminal

1 Configuration

You'll want to start by telling git who you are.

1. Run the following two commands

   ```
   git config --global user.name "My Name"
   git config --global user.email My_email
   ```

   For the first command, keep the quotes, and replace My Name with your name; first and last with a space between is typical. You'll need to use the quotes any time you're providing something with spaces in it, so git knows to treat it all as one string; if there are no spaces you can leave them off or keep them, as you prefer. Now type the second thing but replace `my_email` with your email address. This information will be used to show who made what changes to the code.

2. Troubleshooting:

   - Type the command in yourself. The pdf made the dashes and quotes fancy, but they need to be simple dashes and quotes.
   - On your machine instead of two dashes git bash might only want one.

3. Specify the default name given to the primary *branch* of your code in any repository you create; we will name it main. (sometimes, it is called master)

   ```
   git config --global init.default branch main
   ```

4. Take a screenshot of your terminal/git bash, and add your screenshot to the doc.

2 git init and git status

1. You can use regular Linux commands in the git bash shell. Use `cd` to navigate to the folder that you made for this lab. Once in that folder, you will want to have git set up a repository: `git init` This will create a new subfolder with all the files git needs to track the changes you make to your code. You can't see it but it's there. :)

2. Type

   ```
   git status
   ```

   This tells you which branch you're currently working on, and files in the current folder that are and aren't being tracked by the repository. This is a great command to use regularly to keep you from getting lost.

3. Take a screenshot of your terminal/git bash, and add your screenshot to the doc.

3 git add, git commit, git log

At this point, none of the files are being tracked.

1. Run

   ```
   git add hello.cpp
   ```

   Then run `git status`. The file `hello.cpp` should be listed under a heading "Changes to be committed."

2. Run

   ```
   git commit -m "First commit"
   ```

   This creates an initial save-point for our code. In other words, this creates a snapshot of our code at this moment in time that we can always refer back to in the future. The `-m` means we want to include a message describing the commit. "First commit" is the message, and can be anything you want. You always need to do this, so that when you're looking back you can tell what makes this version of the code different from the previous version.

   If you ever forget the `-m`, your terminal will open a terminal text editor (probably VIM). Exit with the command `:q` + ENTER, which will cancel the commit, and then commit again with the `-m` flag. (if you know how to use VIM, you can type your message inside the editor)

3. Type the following command.

   ```
   git log
   ```

   This lists the past *commits* of your code.

4. Take a screenshot of your terminal/git bash, and add your screenshot to the doc.

4 gitignore, more git add

It is more likely that you will want to add all but a small number of files (for instance, automatically generated files, like `.o` files) to the repository. We will now see how to do that.

1. Create a file in the same folder called `.gitignore` - Note there is nothing before the dot

   - Type `git status`. You should see a list of untracked files under the heading "Changes not staged for commit."
   - Add `myclass.h` of these a single line in the `.gitignore` file and save.
   - Type `git status` again. You should see that `myclass.h` is no longer there listed as being untracked.
   - You can ignore multiple files at one time. Add `*.txt` on a line in your gitignore. Type `git status .` Notice that now all files ending in `*.txt` are no longer listed. Here, * is a wildcard that matches with any string.
   - Comments start after # and run to the end of the line. Add a comment to your gitignore.
   - Take a screenshot of your gitignore and add your screenshot to the doc.

   FYI, you can check out a gitignore file that I copy paste as a starting point for all of my git repositories. Once you've specified which files to ignore, you can add all of the other files in the directory.

2. Type

   ```
   git add .
   ```

   And then

   ```
   git commit -m "Add all files"
   ```

3. Take a screenshot of your terminal/git bash, and add your screenshot to the doc.

4. Take a screenshot of your gitignore, and add your screenshot to the doc.

5 git diff

1. Make some changes to one of your files and save it. Then type

   ```
   git diff
   ```

   This shows us what's different between current files and most recently committed version. You'll see + before lines you added since the last commit, and - before lines deleted since the last commit.

2. Take a screenshot of your terminal/git bash, and add your screenshot to the doc.

6 More on git add

One thing we've glossed over so far is that there are always three different "places" where our files reside:

- working
- staging
- committed

When you modify a file, it automatically moves to working files. The command `git add` moves a working file to staging. By default, only files on staging will be committed by the commit command. This gives you the ability to commit some files but not others, by controlling which files are on staging.

Git will allow you to skip the step of moving files to staging before committing if you want to. To commit all files in working files AND staging, use the -a flag on a commit (this must have just one -):

```
git commit -a -m "description"
```

1. At this point, `hello.cpp`, `myclass.cpp`, and `myclass.h` should all be commited. Change all of them and make another commit with

   ```
   git commit -a -m "description"
   ```

2. Take a screenshot, and add your screenshot to the doc.

7 Branches

As we've mentioned, if you want to make big changes to your code, like adding a new feature, it's good to make a personal copy of the code (or branch) to work in, then merge your changes back into the main branch when you're done. This will help especially when you are working with other people and it's possible you might both modify some of the same files.

1. Create a new branch with the following command:

   ```
   git branch mybranch
   ```

2. Use the following command to switch the branch to your new branch

   ```
   git switch mybranch
   ```

   Once you are on a different branch, if you commit you will not modify the main branch, only your current branch.

3. Make some changes to `hello.cpp` and commit them (remember you need to `git add` then `git commit`, or use `git commit -a`).

4. Switch back to the main branch. The changes you made to `hello.cpp` should no longer be there (you may need to close and reopen the file to see this). Run `git log`, and you should see the commit you made is not there.

5. Now type:

   ```
   git merge -m "message" mybranch
   ```

   The changes you made in `hello.cpp` on `mybranch` should be there again, and if you run `git log`, the commit you made should be there as well.

6. However, merging does not always go so smoothly, and we get a *merge conflict*. Let's see what could go wrong and how to deal with it.

   (a) While still on the main branch, make some changes to `hello.cpp` and add and commit those changes.

   (b) Then switch to the `mybranch` make changes to `hello.cpp` *at the same line(s) of code*, and commit those changes.

   (c) Switch back to the main branch and run

   ```
   git merge -m "message" mybranch
   ```

   You should get an error. Now open `hello.cpp` You should see something like:

   ```
   <<<<<<< HEAD
   edits in main branch
   =======
   edits in branchname branch
   >>>>>>> branchname
   ```

   If you didn't get an error: To create a merge conflict, you need to edit *the same file* in *the same location* on the two branches. If the edits are in different locations of the same file, git is smart enough to resolve the changes on its own.

   (d) Make changes to get the file the way you want it, making sure to get rid of the `<<<<<<<HEAD, =====`, and `>>>>>>>mybranch` that git added. Save your file, add and commit the changes.

   (e) Once you are happy with your merge, delete the branch using the -d flag:

   ```
   git branch -d branchname
   ```

   (f) Take a screenshot and add it to your word doc.

8 Extra Credit. Put your code on Github

1. Create an account on www.github.com if you don't have one already. Create a new public repository on github, add your github repository as a "remote" and push. For credit, need to take a picture of successfully running git push from the terminal, and a picture of your repository on `github.com`. Since this is extra credit, I will let you figure this out on your own.