Week 08 Lab Exercise

Git Basics

Objectives

- to learn the basics of Git and GitHub
- to understand how Git commands work internally

Admin

If you haven't used Git or GitHub before, this lab will help you get started with the basics, so that you can begin contributing code to your assignment. If you are already familiar with Git, some of the content may still be new to you, so you might want to scan through the lab anyway. **Please note that Exercise 2 is compulsory.**

The purpose of this lab is solely to help you use Git and GitHub in Assignment 2 - you won't be assessed on any of this.

Background

Git is a version-control system, which enables developers to track changes during software development. In short, it keeps track of snapshots of your code during development. Meanwhile, GitHub is a web interface to Git that includes many of its own features. It allows developers working in different locations (and on separate machines) to collaborate by acting as a central hub to/from which they push and pull code, and enables them to keep up to date with the most recent version of the code.

Setting Up

You should have already created a GitHub account for working on Assignment 2. If you haven't created a GitHub account yet, sign up here, and make sure that the owner of your group's Assignment 2 repository invites you to the repository as a collaborator.

Exercise 1 - Git Basics

Creating a Repository

The first step in this lab is to create a GitHub repository, which we will use to demonstrate the basic Git commands. A repository is like a folder where you store all the code related to your project. To create a repository on GitHub, make sure you are signed in, and then go to this page. Then:

1. Name the repository **cs2521-lab08**
2. Select **Private** to make the repository private
3. Check the box next to **Initialize this repository with a README**
4. Select **C** in the **Add .gitignore** dropdown
5. Click on **Create repository**

Cloning the Repository

In the previous step, you created a repository on GitHub. To make changes to the repo and share your changes with others, you first need to make a local copy of it. This is called **cloning**. To clone the repo to your local machine, first go to the main page of the repo that you created above, click on the green button which says **Code**, and then copy the URL in the box that appears. Then, in your terminal, run the command:

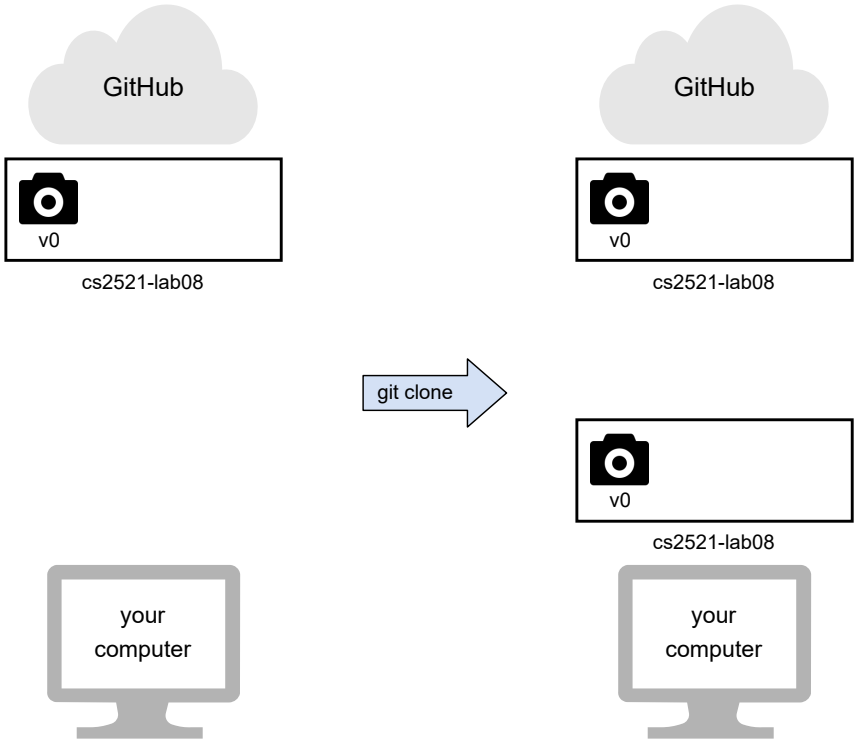
```
$ git clone repository-url
Cloning into 'cs2521-lab08'...
Username for 'https://github.com': your-username
Password for 'https://your-username@github.com':
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
```

where *repository-url* is the URL that you just copied from GitHub. Enter your GitHub username and password if you are prompted.

Now you have a local copy of your repository. Change into the repository directory (i.e., the directory that was created when you ran **`git clone`**) and compare the contents of the directory with the contents of the repository on GitHub. Are they the same? (They should be.)

```
$ cd cs2521-lab08
$ ls
README.md
$ cat README.md
# cs2521-lab08
```

The diagram below depicts what happened when you ran **`git clone`**. Note that the repository on GitHub (called the remote) and the repository on your local machine (called the local repository) are separate versions of the repository - if the remote repository is updated, the changes won't be reflected in your local repository until you *pull* from it, and if you update your local repository, the changes won't be reflected in the remote repository (on GitHub) until you *push* to it.



The camera icon represents a commit (which is essentially a snapshot of your repository) - when you created the repository on GitHub, GitHub created an initial commit (containing `README.md` and `.gitignore`) that you then retrieved when you ran **`git clone`**.

Making a Commit

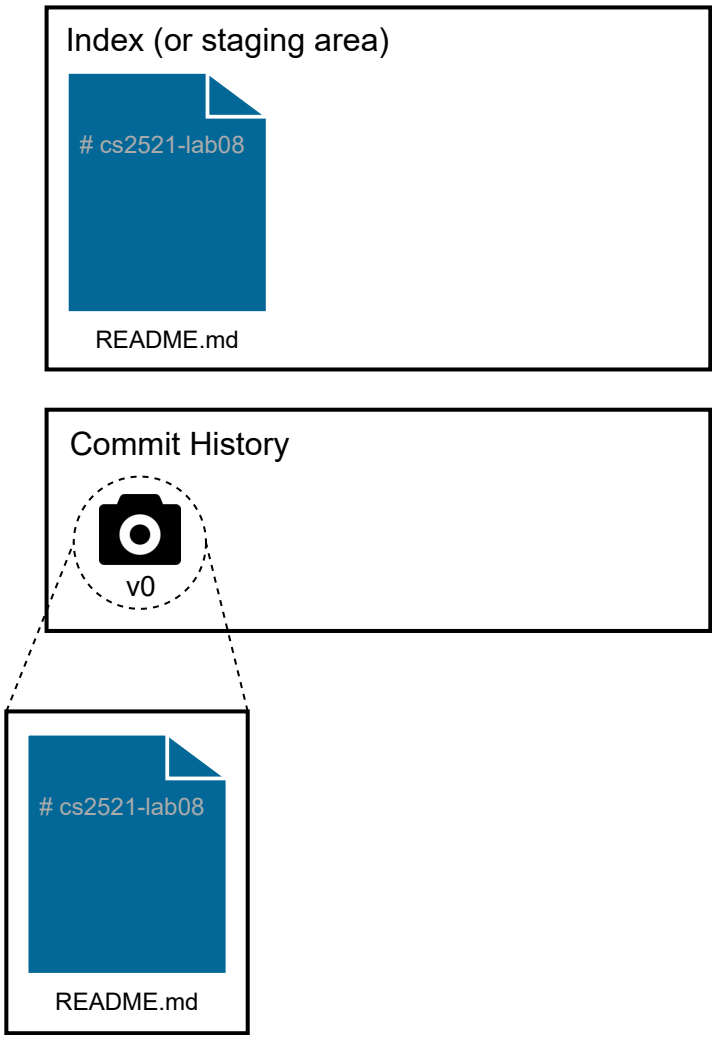
Now that you have cloned the repo, you are ready to work on the codebase locally. Let's make some small changes for demonstration purposes. Create a file called `helloWorld.c` and add a bit of code to it. Also add a line (containing anything you want) to `README.md`.

Suppose you now want to *push* your changes to GitHub. Before you do this, you need to make a *commit*, which is basically a snapshot of the repository (technically, it's a snapshot of changes to the repository).

Before you can commit, you have to do what is called staging your changes, which effectively tells Git what changes you actually want to commit. This is where a little bit more detail about how Git works behind the scenes will be helpful.

Detour: Basics of Git Internals

A Git repository is composed of two main parts that are relevant to us - the **index** and the **commit history**. The index, also known as the staging area, is where you assemble your files to make commits, which are snapshots of changes to the index at various points in time. Meanwhile, the commit history is where the commits are actually stored. This is the current state of your local Git repository:



(If you're wondering where all of this is actually stored, it's in a hidden directory called `.git`. You can poke around in there, but don't change anything.)

Here are some observations:

- `v0` is the initial commit (snapshot) of the repository. It contains only `README.md`, which makes sense because when you first created your repository on GitHub, it consisted of only that file (we'll ignore `.gitignore`).
- The index contains `README.md`, but not `helloWorld.c`, which is the file that you just created. Also, `README.md` in the index does not contain the extra line that you added. This is because you haven't added your changes to the index.

You can actually see and compare the status of the files in your working directory and the index by using the **`git status`** command:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
    modified:   README.md

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    helloWorld.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Don't worry about branches for now. The output gives us some very useful pieces of information:

- `README.md` has been modified, but its changes are not staged for commit. This means we've modified `README.md`, but haven't added the changes to the index.
- `helloWorld.c` is untracked. This means that `helloWorld.c` is in the working directory, but not in the index.
- No changes have been added to the commit. This means that the state of the index hasn't changed since the last commit, which means there would be no point in making a commit.

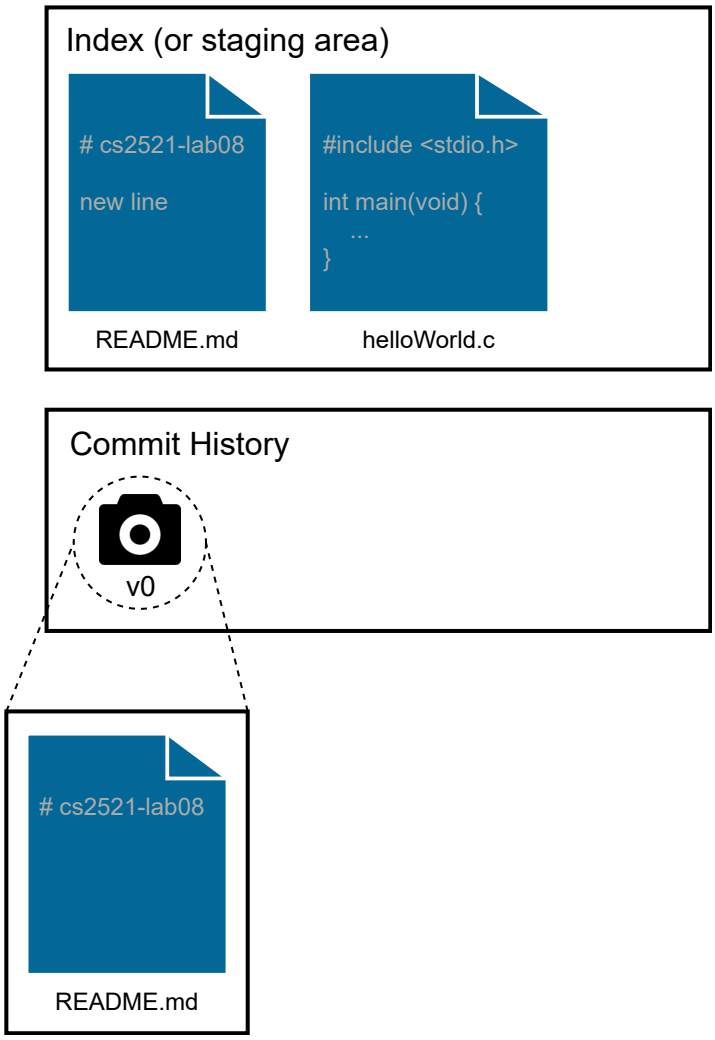
To add your changes and new files to the index, you can use the **`git add`** command. You can either add the files individually:

```
$ git add helloWorld.c
$ git add README.md
```

...or you can add all of the changes at once:

```
$ git add -A
```

Now, your changes will be added to the index, like so:



Note that `v0` doesn't change, as it is a snapshot of the index at a previous point in time - you can't change it.

Now, you are ready to make the commit, that is, take a snapshot of the index. But before you do that, let's run **`git status`** again to see how the state of the index has changed:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD ..." to unstage)

    modified:   README.md
    new file:   helloWorld.c
```

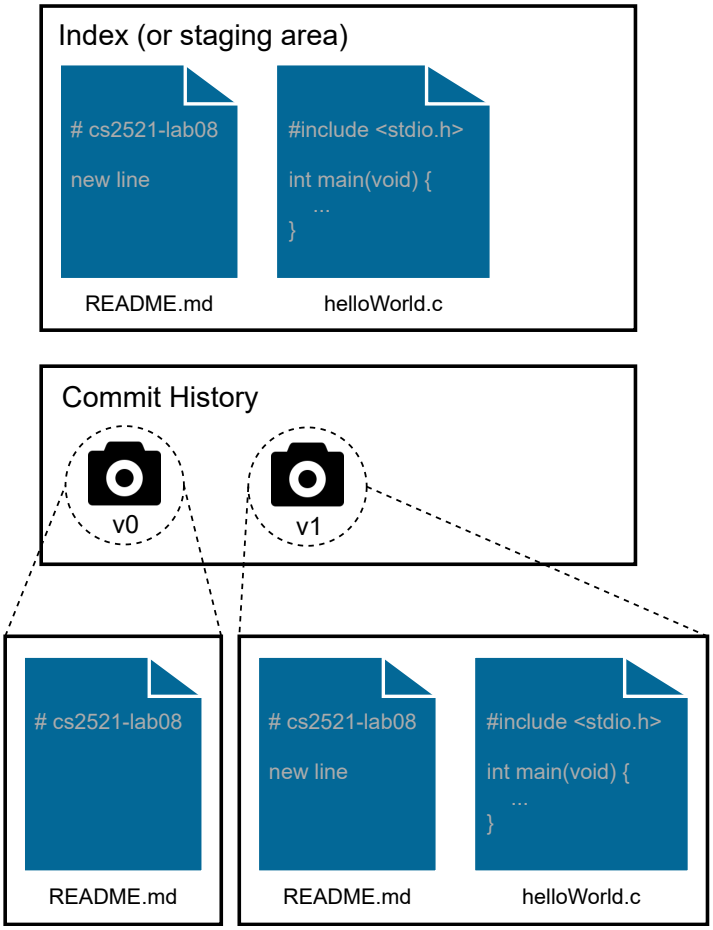
Notice that **`git status`** now reports that the modifications to `README.md` and the new file `helloWorld.c` are ready to be committed. This emphasises the fact that updating a Git repository is a two-step process: you first add the changes to the index (the staging area), and then make a commit.

The command to make a commit is **`git commit -m "`*`commit message`*`"`**. For example:

```
$ git commit -m "added helloWorld.c"
[master 997b345] added helloWorld.c
 2 files changed, 10 insertions(+), 1 deletion(-)
 create mode 100644 helloWorld.c
```

IMPORTANT: You should always include a descriptive commit message with every commit that you make, so that other developers can tell what changes you've made.

This saves the current state of the index in the commit history, so we now have a new commit:



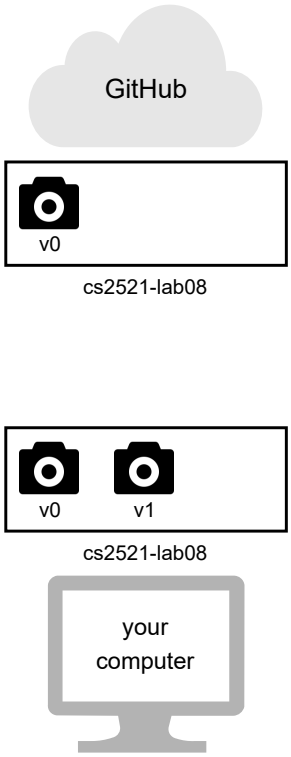
Congratulations! You've now made a commit. Every time you finish part of a project/assignment and get it working, you should make a commit, since it saves a version of the code that you can fall back on (in case something goes wrong later), and you can now push the code to GitHub, so that your teammates can access it. If you run **`git status`** again, you'll see that there is now nothing to commit, since you have just committed your most recent changes.

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Pushing

So you've successfully made a commit, but this commit isn't visible on GitHub yet. In fact, if you check your repository on GitHub, you'll see that it still only contains `README.md`, and `README.md` doesn't even contain the new line that you added. What you need to do is to push your changes to GitHub. Here is the situation:



To push your changes, use the **`git push`** command. This is guaranteed to work, since you are the only person who has access to your repository, but issues can arise if multiple people are working on the same repository (we'll explore these in a bit).

```
$ git push
Username for 'https://github.com': your-username
Password for 'https://your-username@github.com':
Counting objects: 4, done.
Delta compression using up to 12 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 422 bytes | 211.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/your-username/cs2521-lab08.git
   c54fd0e..997b345  master -> master
```

Now, if you refresh GitHub, your changes will be visible, and your commit message will appear next to each file that you modified/added.

Well done! You've now learned how to clone a GitHub repository, add changes, make commits, and push to GitHub.

Pulling

Usually, when you are using Git, it is in a team. That means that you will not be the only one who is making the changes. If someone else makes a change and pushes it to the server, your local repo will not have the most up-to-date version of the files. Luckily, Git makes it easy to update your local copy with the **`git pull`** command.

`git pull` connects to the remote repository that your local repository is linked to and checks that all of your files are up to date. This ensures that you don't accidentally do things like implement the same thing someone else has already done and also lets you use other people's work (e.g., new functions) when developing.

Pulling regularly is one of the **most important** practices in Git!

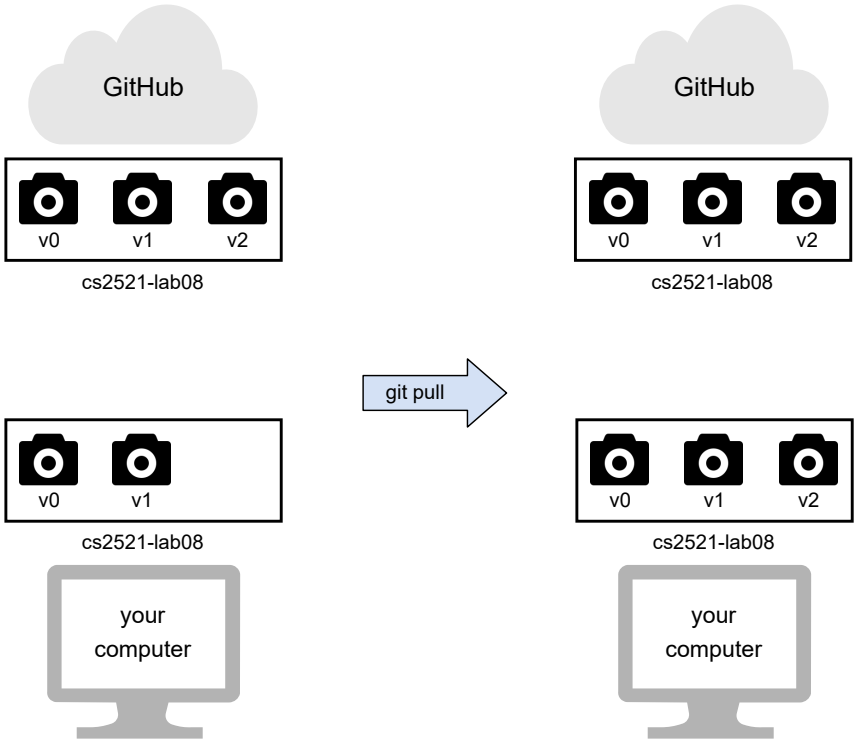
Unfortunately, in this lab you are just working individually. But GitHub still gives us a nice way to demonstrate **`git pull`**.

Let's make a simple change in the GitHub repo to demonstrate pulling. On the main page of your GitHub repository, click on `README.md` to view the file, and then click on the pencil (edit) icon on the right-hand side. Add another line to the file, and then click on **Commit changes** at the bottom of the page.

We've now simulated what happens when one of your teammates pushes a change to GitHub. The changes are visible on GitHub, but you don't have them yet. To obtain the new changes, go back to your terminal and use the **`git pull`** command:

```
$ git pull
Username for 'https://github.com': your-username
Password for 'https://your-username@github.com':
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/your-username/cs2521-lab08
   997b345..bfff7f9  master     -> origin/master
Updating 997b345..bfff7f9
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
```

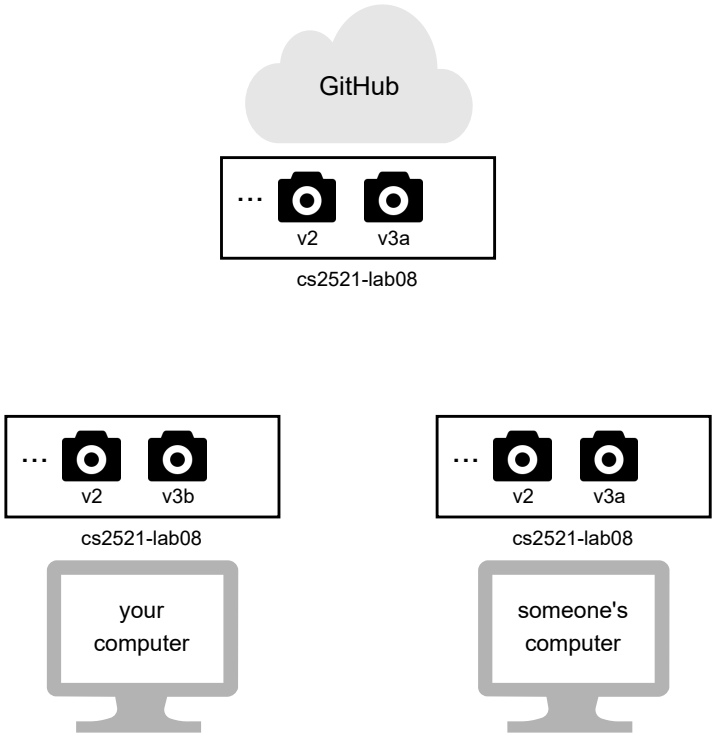
If you now open the `README.md` file, you should find that it now contains the new line that you added. Well done! Now you know how to keep your local repository up to date.



Problems with Pushing

Pushing will not always work, especially if multiple people are working on the same repository or if you do your work on multiple machines.

One problem that can occur is when you try to push you work to GitHub, but someone else has pushed a change before you.



You can simulate this scenario by doing the following:

1. On GitHub, modify `README.md` in the same way that you did in the **Pulling** section above, and then click **Commit changes**.
2. In your local repo, modify `helloWorld.c`. Then, add the changes using **`git add`**, commit them using **`git commit`**, and then use **`git push`**.

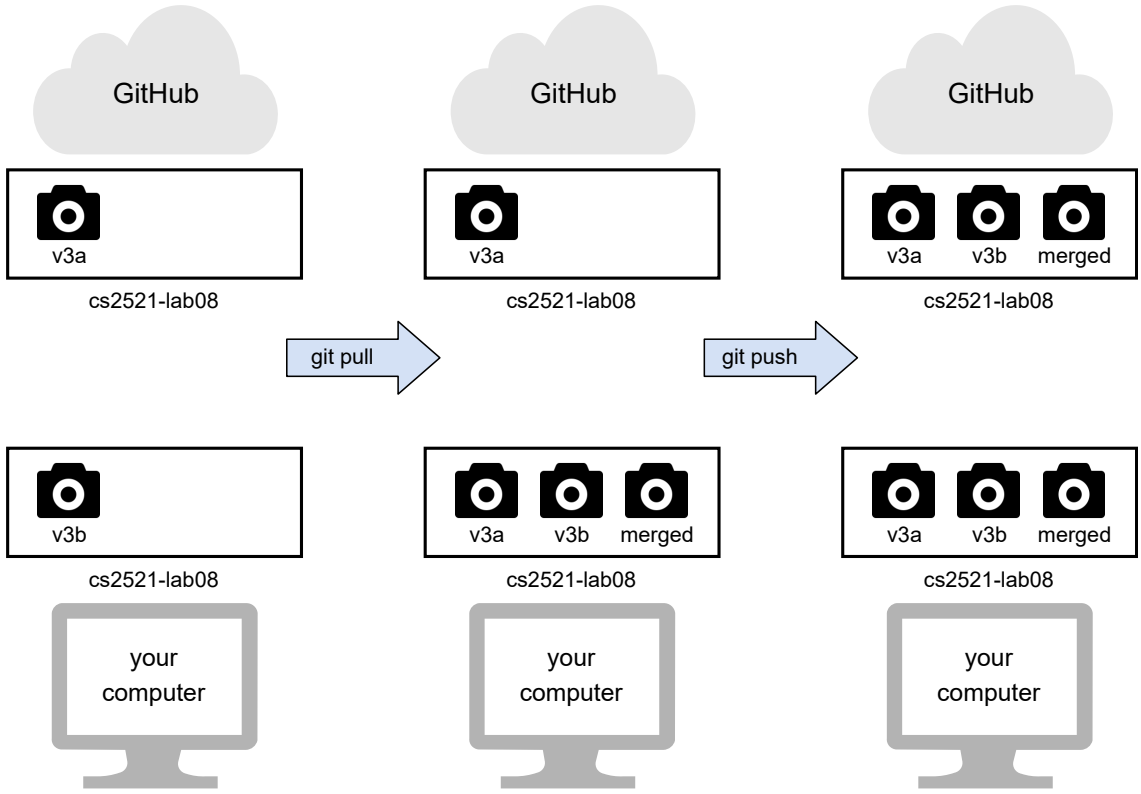
You should get a message that looks like this:

```
$ git push
Username for 'https://github.com': your-username
Password for 'https://your-username@github.com':
To https://github.com/your-username/cs2521-lab08.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/your-username/cs2521-lab08.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

To fix this, you should first use **`git pull`** to integrate the changes on GitHub with your changes, as the hint suggests. As long as the changes don't conflict, **`git pull`** will merge the changes on GitHub with the changes that you have made (you may have to write a commit message for the merge operation), and then you can safely push.

```
$ git pull
...
From https://github.com/your-username/cs2521-lab08
   bfff7f9..1441789  master     -> origin/master
Merge made by the 'recursive' strategy.
 README.md | 2 ++
 1 file changed, 2 insertions(+)

$ git push
...
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/your-username/cs2521-lab08.git
   1441789..3c13e17  master -> master
```



Problems with Pulling

You can also get issues with pulling. One problem that can occur is when you try to pull a commit from GitHub, but that commit involves a change to a file that you have modified locally (and you haven't committed the changes).

You can also simulate this scenario by doing the following:

1. On GitHub, modify `helloWorld.c` by adding a comment at the top of the file, and then click **Commit changes**.
2. In your local repo, modify `helloWorld.c` by adding a comment at the bottom of the file. Do NOT commit this. Then, use **`git pull`**.

You should get a message that looks like this:

```
$ git pull
...
From https://github.com/your-username/cs2521-lab08
   3c13e17..eadeeda  master     -> origin/master
Updating 3c13e17..eadeeda
error: Your local changes to the following files would be overwritten by merge:
    helloWorld.c
Please commit your changes or stash them before you merge.
Aborting
```

To fix this, you should first add and commit your local changes, as the message suggests, and then use **`git pull`**.

```
$ git add -A
$ git commit -m "added footer comment"
[master cea168a] added footer comment
 1 file changed, 1 insertion(+)
$ git pull
...
Auto-merging helloWorld.c
Merge made by the 'recursive' strategy.
 helloWorld.c | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Notice that the output mentions 'auto-merging'. This occurs when the changes from GitHub and the local changes both contain changes to the same file, but Git is able to sort them out and merge them. If Git is able to do this, this usually means the changes occurred in different parts of the file (which is why we asked you to modify the top and bottom of `helloWorld.c` earlier). However, if you and your teammate both change the same part of the file, then a merge conflict can occur.

Merge Conflicts

Merge conflicts are the one necessary downside to Git. Luckily, they can be avoided most of the time through good use of techniques like regular commits, pushes and pulls. Merge conflicts occur when Git cannot work out which particular change to a file you really want.

To engineer a merge conflict:

1. On GitHub, add a line to the end of `README.md` (containing anything you want), and commit the change.
2. In your local repo, add a different line to the end of `README.md`. Then add and commit these changes.
3. Use **`git pull`**.

You should get output that looks like this:

```
$ git pull
...
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

If you open `README.md`, you will see some additional syntax that Git has added to indicate the merge conflict(s). (Note that there may be multiple merge conflicts in the same file - in this case, there is only one.)

```
$ cat README.md
# cs2521-lab08

...
```

```
<<<<<<< HEAD
goodbye
=======
hello
>>>>>>> f670d1c6e3bb70e24ece382e4aba865e0d59427f
```

- `<<<<<<<` marks the beginning of the conflicting **local** changes.
- `=======` marks the beginning of the conflicting **incoming** changes.
- `>>>>>>>` marks the end of the conflict zone.

Resolving a merge conflict is as simple as editing the file normally, and choosing what you want to keep in the places Git wasn't sure.

The above conflict can be solved in many ways. One way would be to keep both changes. Once we have decided on this we just need to remove the additional syntax. The resolved file would be as follows:

```
$ cat README.md
# cs2521-lab08

...

goodbye
hello
```

After you have fixed all of the conflicts in the file, add and commit the resolved file, and the merge conflict is resolved!

Exercise 2 - Assignment 2 README.md

Use your knowledge of Git to add your GitHub username, real name, zID, and tutorial code to the `README.md` file in your Assignment 2 repository (all group members should do this). Also add your group name to the `README.md` file (only one person from the group needs to do this).
