Basic - Git 1

EECS 201 Winter 2025

Submission Instructions

This homework will be submitted as a repository on the EECS GitLab server. This will become evident as you work through the assignment.

To finish the submission process and get your grade, you will need to run the autograder. You will need to SSH into the course server (see Basic - Intro):
`local$ ssh uniqname@peritia.eecs.umich.edu`
(the `local$` referring to a shell prompt on your system)

and then run `eecs201-test`:
`peritia$ eecs201-test basic-git1`
(the `peritia$` referring to a shell prompt on the server)

Preface

Git has an interesting conundrum where to use Git you need to have some understanding of Git, but to understand Git you need to have some experience using Git. While this class' goal is to help you become more independent problem solvers, this circular dependency may make Git hard to get into, so for this homework we'll give a guided introduction.

Some other tutorials/resources you can look at besides the official documentation and *Pro Git*:

- Atlassian's tutorials
  Atlassian is the company that made BitBucket, JIRA, and Confluence
- Interactive branching game
- A humorous quick-reference guide
- A more humorous quick-reference guide (language warning)

In this introduction we will be taking an existing codebase and turning it into a Git repository. The process of doing this in an empty directory is exactly the same: instead of an empty directory we have untracked files that are already provided.

1 GITing started

You are in charge of migrating a small command line calculator program from an archaic, proprietary version control system, whose company became defunct in the 90s, to Git as well as handling the remaining issues in the code.

1. First, we'll need to acquire the code. The URL for the initial files is `https://www.eecs.umich.edu/courses/eecs201/wn2025/files/assignments/basic-git1.tar.gz`. **Feel free to use your browser to open this link and download the file and once you do, go to step 2.**

   If you want to do this from the command line, you can as well! However, note that due to some changes in certificates the process can be a bit more involved than it was in previous semesters. I'm providing these for posterity, or if you really want to do this from the command line.
   WARNING: recently the certificates used by UMich (and other schools) got renewed, which don't seem to be present in a lot of system-provided certificate stores and bundles (but are in that of browser's). I will present both the insecure, workaround way and a more secure workaround.

There are two tools we can use to do this from the command line. The one popular option is `wget`, which allows you to download files through HTTP and FTP. Unfortunately, macOS does not have easy access to `wget` without Homebrew. That leads us to the other option `curl` which is a general tool for making transactions with a server. `curl` is much more powerful as a general tool. For the following, the `#` represent comments: you don't need to run them.

If you want to use `wget`:

```
# The --no-check-certificate tells it to ignore checking certificates
$ wget --no-check-certificate https://www.eecs.umich.edu/courses/eecs201/wn2025/files/assignments/basic-git1.tar.gz
```

If you want to use `curl`:

```
# That's the capital letter O, not the number 0
# The -k tells it to ignore checking certificates
$ curl -k -O https://www.eecs.umich.edu/courses/eecs201/wn2025/files/assignments/basic-git1.tar.gz
```

If you actually want a more secure option with the command line, you can download this file provided by ITS and then supply it as an argument for `wget` and `curl`:

```
$ wget --ca-certificate=/path/to/downloaded/certificate
       https://www.eecs.umich.edu/courses/eecs201/wn2025/files/assignments/basic-git1.tar.gz
$ curl --cacert=/path/to/downloaded/certificate -O
       https://www.eecs.umich.edu/courses/eecs201/wn2025/files/assignments/basic-git1.tar.gz
```

2. You should now see a file called `basic-git1.tar.gz` in your current directory (or wherever you downloaded it to). The file is a zipped (compressed via gzip, hence `.gz`) tarball (hence `.tar`), which is the result of `tar`. `tar` is an archiving utility, which fulfills a similar purpose as WinRAR, 7zip, etc. We can decompress and extract it (i.e. "unzipping" it, using zip parlance). You can run either of the listed `tar` commands to do so.

```
$ # e(X)tract g(Z)ipped (F)ile
$ tar xzf basic-git1.tar.gz

$ # most current versions of tar do autodetection and don't actually need the z flag
$ # in the future you can just omit the z flag: I just left it in for posterity and completeness
$ tar xf basic-git1.tar.gz
```

   If you get an error and you used `wget` or `curl` chances are you made a typo with your commands and the server gave you a 404 Not Found webpage instead of the desired file.

3. `cd` into the extracted `basic-git1/calculator` directory.

4. Let's play around a bit with the application. Run `$ make` to build the application. You may see some compiler **warnings**: these are fine and are because of the code, not you, and are intentional. Familiarize yourself with the source code in `src` and `inc`.

5. Try running `$ ./calculate 3 + 3`

6. Try running `$ ./calculate 3 x 3` (that's the letter x, not an asterisk)
   The application error is intended: the code has missing features that you'll be adding. We haven't gotten into it yet, but you can directly run *executable* files by specifying their path. If they're in the current directory, as a safety mechanism you'll have to have a slash in the path hence the `./` Feel free to run other operations.

7. Let's git started. Are you inside the `basic-git1/calculator` directory? Make sure you're there!
   Initialize a Git repository with `$ git init -b master`. The `-b` flag specifies the name of the initial branch. This is the default on many systems, but isn't necessarily mandatory. For this assignment we'll be using `master`, but other assignments will have different branch names. Branch names are aribtray and have meaning only for the developers: Git is not programmed to recognize `master` or any other name as special.

8. Let's blindly add all the files in the current directory with `$ git add .` **DO NOT commit just yet!**
   Note I said "blindly". What this does is add *everything* in the current directory (`.`). In most cases, you don't really want to add *everything*; you usually want to be a bit more selective on what you want to put into each commit. However, when it comes to the initial steps of setting up a repo, it may be useful: just be judicious in what you commit (we'll touch on this later). **Again, DO NOT commit just yet!**

9. Run `$ git status` to see what's currently in the Index (get into the habit of running `git status`; it'll tell you a lot about what's going on).

10. Notice how we have the final compiled binary `calculate` and intermediate object code files in `obj` in the Index. In general, we don't want to version the intermediate and final build outputs as they're the products of the code that we are versioning; no reason to version a thing twice. Other things that we don't tend to version include development environment specific things, like logs, other output files, or core dumps that your application produces during runtime, and developer system dependent things like editor swap files and weird file system helper files like `.DS_STORE`.

11. Use `$ git reset calculate` to unstage the `calculate` binary. Now use `git reset` to unstage files under `obj/` (you can specify the directory itself to get them all in one fell swoop). As a note, `git reset` can reset multiple things if you provide more files/directories arguments e.g. `git reset file1 file2 dir3`. If your OS or file system creates any other junk specific to it (like `.DS_STORE` on macOS), unstage those as well.

12. Before you commit, you may want to set the text editor used for Git commit messages. You can change the `core.editor` Git variable to do this. For example, if I wanted to set the editor to `nano` globally (i.e. with all repos on your system unless a repo specifically overrides it): `$ git config --global core.editor nano`. The Pro Git book has a section with some commands to configure Git to use various common editors.

13. Run `$ git status`. Make sure that only source code files are staged: `calculate` and `obj/` should not be staged, and if you're on macOS neither should `.DS_STORE` and `.dSYM`.

14. Use `git commit` to make this initial check-in/commit of the codebase. While you should make a commit message that follows best practices as mentioned in class, the first one can be a bit weird: a common message is simply "Initial commit".

15. Run `$ git status` for good measure. Take note of what branch you're on. If you are not on a branch named `master` (either because your version of Git has changed the default branch name, or you have configured the default branch name to be something else), go ahead and run `$ git branch master` to create the branch, and the run `$ git switch master` or `$ git checkout master` to switch over to the `master` branch. We'll be using the `master` branch to submit the assignment. (The branch that'll be graded will change from assignment to assignment).

16. Run `$ git status` again. Isn't seeing `calculate` and `obj/` as untracked files annoying in `git status`? We can get Git to ignore files and directories through the use of a `.gitignore` file. Inside this file you can list out the names of files and directories that you want Git to ignore, one on each line, and can give it some patterns as well e.g. `*` is a wildcard. You can find the details about `.gitignore` in the official documentation. Get Git to locally ignore the intermediate and final build outputs (`calculate` and the stuff under `obj/`). If your OS, file system (or text editor) creates files like `.DS_STORE` or `*.swp` files, you may want to look into a way to ignore such files *globally*. For example `$ nano .gitignore` will open up an instance of `nano` that will save to `.gitignore`.

17. Stage the `.gitignore` file and commit it.

18. Run `$ make clean` to delete the intermediate and final build outputs.

19. Create a *plain-text* `README` file that explains how to build and clean the application. For example `$ nano README` will open up an instance of `nano` that will save to `README`.

20. Stage and commit your `README` file.

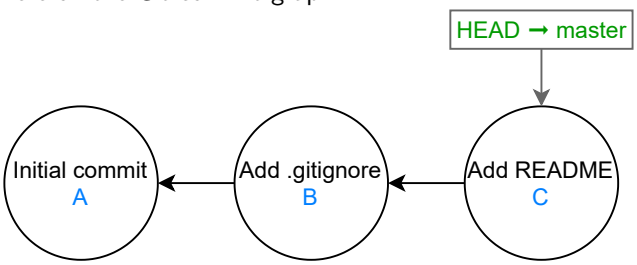
21. `rm` the `Makefile` and delete a character from the `README`.

22. Recover the deleted `Makefile` undo the changes to the `README` using `git restore` or `git checkout`.

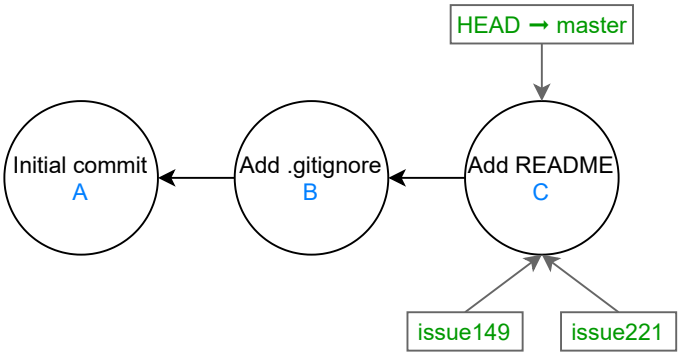
Hooray! We've migrated the application to Git. Now we have to move onto fixing our problems.

2 GITing around

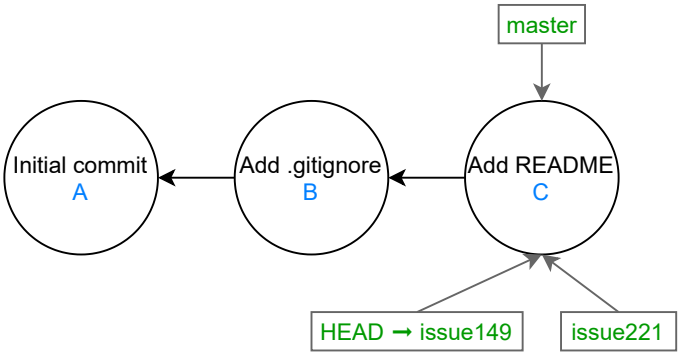
1. Take note of the comments that mention an issue in the source code under `src/`. At this point you should be here on the Git commit graph:



2. From here, we are going to make two *topic branches* that deal with each of the issues. Use `git branch` to create branches `issue149` and `issue221` that correspond to the issues mentioned in the comments.



3. Switch to `issue149` using `git switch` or `git checkout`.



4. Lets fix up its code, implement the fix by adding this case in the `switch` block as follows below the comment menioning the issue by adding this case in the specified file:

```
src/calc.c
---
    case OP_MUL:
        *result = arg1 * arg2;
        return STATUS_OKAY;
```

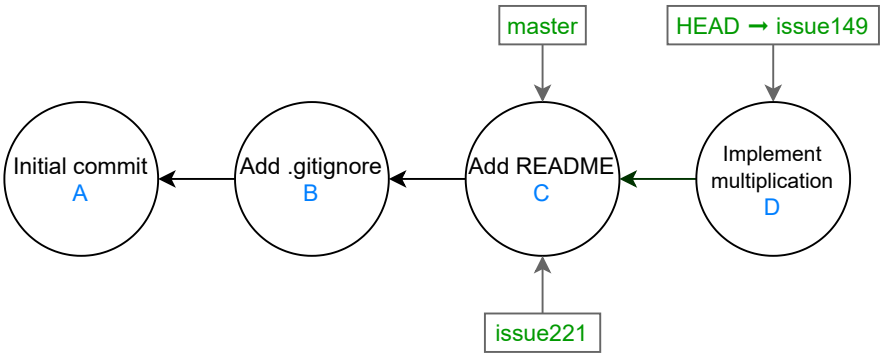
5. **Make sure to delete the comment mentioning the issue in the code, as you have now dealt with it.**

6. **Run `$ make` to compile the application**. Test out `calculate` to see if the issue of the current branch is resolved by performing the multiplication again. Make sure that it can be compiled and run!

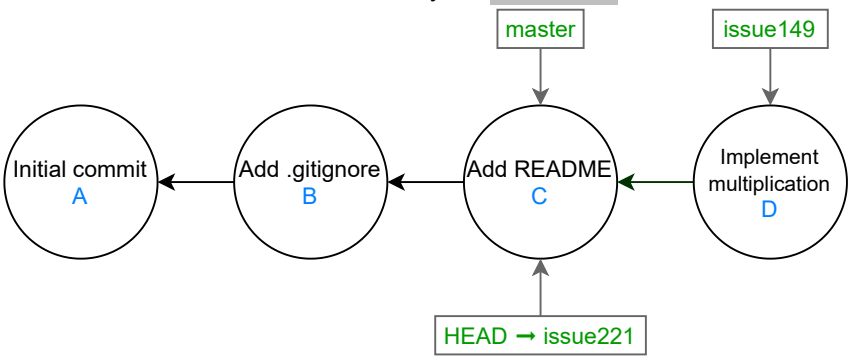
7. Run `$ git diff` to see how your files have changed.

8. Run `$ git add -u` to add files that you have modified. If you have staged files that are unrelated to this issue, be sure to unstage them as this branch is focused on fixing the appropriate issue.

9. Commit the fix, making sure to have a properly formatted and descriptive title and body for the commit message as well as including a reference to the issue number in the **body**. Remember to go by the style guideline!



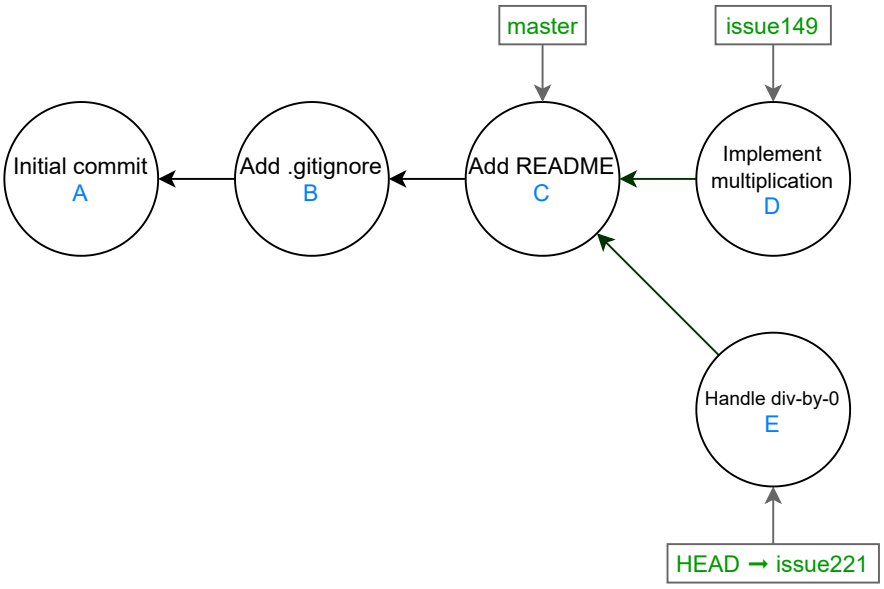
10. Switch to the branch `issue221`. You'll notice that the changes made in branch `issue149` are gone: the commit made for it is not in the history for `issue221`.



Repeat steps 4 through 9, implementing the fix as follows by adding this case:

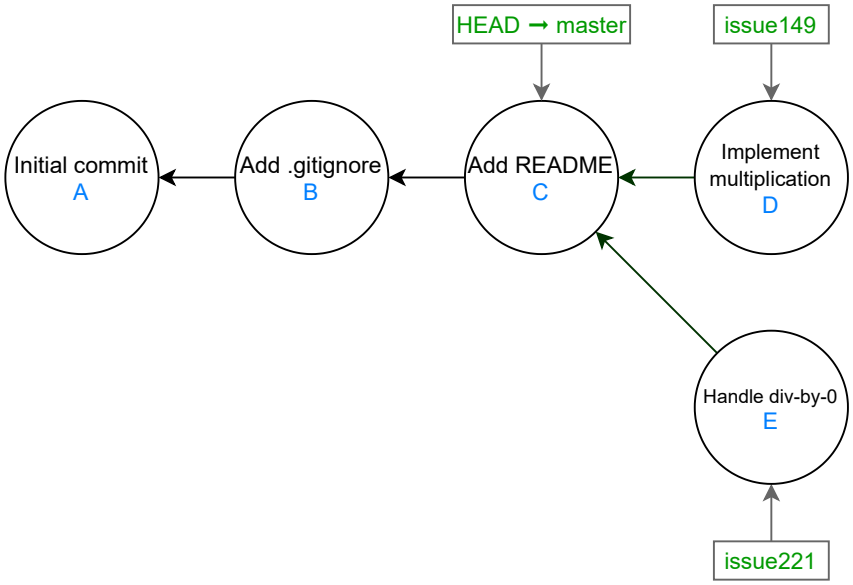
```
src/main.c
---
    case STATUS_DIV_BY_ZERO:
        fprintf(stderr, "Divide by Zero\n");
        return 5;


src/calc.c
---
        if (arg2 == 0) return STATUS_DIV_BY_ZERO;
```



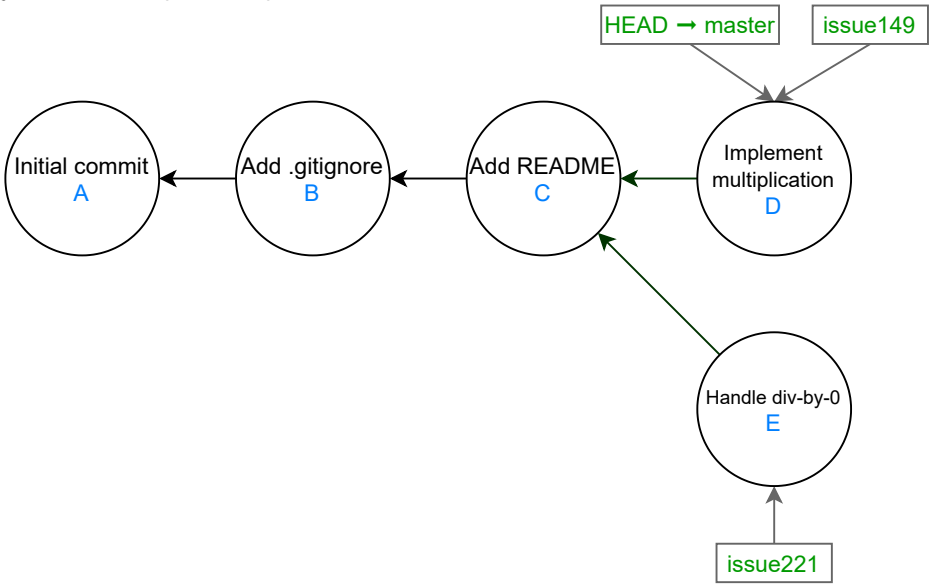
11. Compile it again with `$ make`. Make sure there are no compilation errors (warnings are fine)!
12. Test to see that `calculate` can handle a divide by zero: `$ ./calculate 3 / 0`.
13. Make sure that you've made your commit for `issue221`.

14. Switch to the `master` branch.

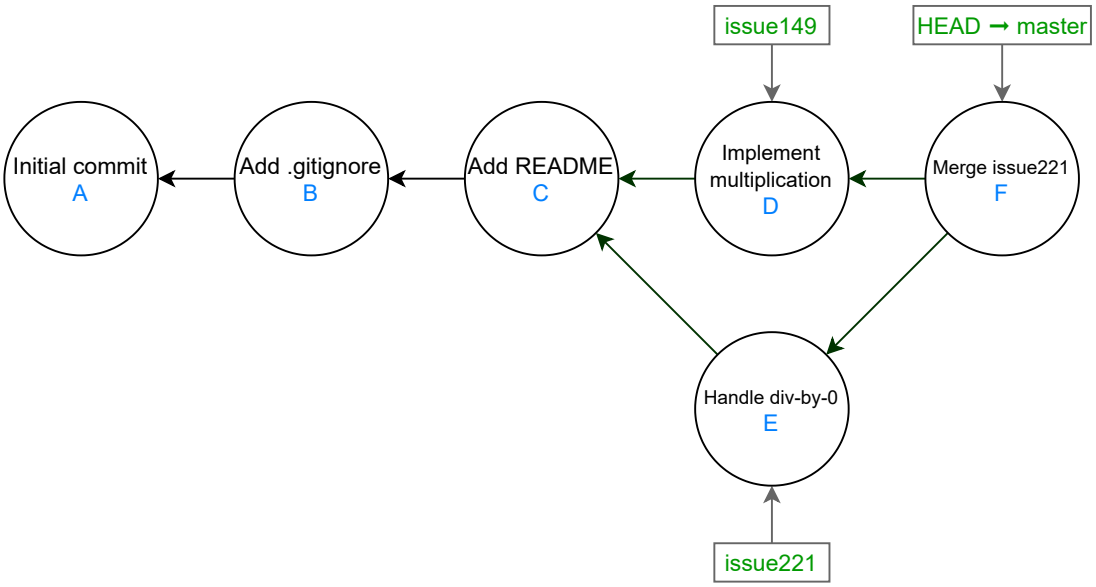


15. Now we are going to bring the commits from the topic branches over to the `master` branch.

16. Run `$ git merge issue149` to bring over the commits from the `issue149` branch. Note that `issue149` and `master` **have not** diverged, making this process painless. This is known as *fast-forwarding*, where a branch just moves its pointer up to where another branch is.



17. Now use `git merge` to bring over the commits from `issue221`. Note that `issue221` and `master` **have** now diverged with the topic branch since `master` has moved forward to where `issue149` is while `issue221` had sprouted off from `master`'s old position. This will necessitate a special *merge commit* that gets automatically generated (which you don't have to reword for this assignment). If the merge has to stop due to not being automatically resolved, use `git status` to see where the merge conflict is occurring and modify the files to get them into working order (i.e. it compiles and runs as intended for that branch), then complete the merge (`git status` will give you instructions on how to do this).



18. Run `$ git show` to see what this latest commit entails.
19. Run `$ git show HEAD^1` or `$ git show HEAD^` to see the first parent of this merge commit.
20. Run `$ git show HEAD^2` to see the second parent of this merge commit.
21. Run `$ git log` to show you the list of commits. If `git log` has used a "terminal pager" (e.g. it gives you an interface that lets you scroll up and down the log, and has a `:` at the bottom left), you can exit out of it by hitting `q`.

Now that your job is finished, let's push it to a remote repository.

3 You're going surfing on the internet!

1. Log into https://gitlab.eecs.umich.edu and set up your EECS GitLab account.
2. Set up SSH with your EECS GitLab account (if you haven't already). Remember that SSH key you created in Basic - Intro? It'll come to play here. You can find this setting under your account settings > SSH Keys, and the site has a guide on how to set it up. This allows you to painlessly clone/pull/push with the EECS GitLab server. (The counterpart to SSH is HTTPS, but SSH is seriously easier to work with on a regular basis).
3. Create a completely blank, without a README, new **private** "Project" (i.e. remote repository) with exactly the name: `eecs201-basic-git1`. **Remember to uncheck the box about creating a README!**
   GitLab "Projects" contain more than just a Git remote repository: they also have things such as membership management, issue tracking, etc.
4. Let's now set up this EECS GitLab Project as the remote for your local repository.
5. Add it as the "remote" named `origin`:
   `$ git remote add origin git@gitlab.eecs.umich.edu:<your uniqname>/eecs201-basic-git1.git`.
   e.g. for me:
   `$ git remote add origin git@gitlab.eecs.umich.edu:brng/eecs201-basic-git1.git`
   This uses `git remote` to add a "remote" named `origin` with a specified SSH URL. A "remote" is another place that the repo is stored: in this case, we're using the EECS GitLab server as a place to store a "remote" copy. `origin` is often used as a name for the main place the code is remotely stored, and this name used for this local repo as a shorthand for the entire URL.
6. Now let's push the all commits from all the branches to the remote:
   `$ git push -u origin --all`.
   `-u` sets the upstream tracking information for the local branches, allowing them to push/pull commits from the remote branches.
   `origin` is the remote that we are referring to.
   `--all` pushes all of the branches. You could instead, say, push only `master` e.g. `git push -u origin master`.
   You only need the `-u` flag when you do your first push: this flag is there to set the initial information about *where* the code is going. On subsequent runs of `git push` it will remember where the push is going.
7. In the EECS GitLab Project's Manage>Members>Invite members and add `brng` as a **Reporter**. As part of the grading process for this assignment we will be looking at your repositories on the EECS GitLab.
8. Back in your local repository, make sure you are in `master`.
9. Create a file called `report.txt`.
10. On the first line of the file, put down an estimate of how long you took to do this assignment in minutes as an **integer (e.g. "37", "84": just numbers, no letters)**.
11. On the second line and onwards of the file, put down what you learned (if anything) by doing this assignment. **If you already knew how to do all of this, just put "N/A"**.
12. Stage and commit this `report.txt` on `master`.
13. Run `git push` to push this commit to your EECS GitLab repository.

4 Autograder

The course server has the ability to grade your assignment and give you credit! **This will be done for all assignments submitted via the EECS GitLab server**. Let's trying running it for this assignment:

1. If you haven't already, fill out the course server account form (see Basic - Intro).
2. Log in to the course server. If you don't remember how to, you can run `$ ssh <youruniqname>@peritia.eecs.umich.edu` e.g. **for me**, I'd run `$ ssh brng@peritia.eecs.umich.edu`.
3. Run `$ eecs201-test`: by itself it'll print out usage instructions as well as the gradable assignments at that given moment.
4. Run `$ eecs201-test basic-git1` to run the autograder on your assignment.

   Please note that there is a cooldown period between runs: this is meant to encourage you to take a moment and evaluate what you have done locally. In many professional situations shared testing resources are in high demand, so your code might not get to be tested immediately, and even then the tests may take many minutes or even hours to complete. It's generally good form to do some local sanity checks (e.g. checking if the code/submission meets some basic requirements such as being able compile and run without immediately running into an error, or running some basic tests) so you don't wait several hours to find out you had a compile error or that your application immediately crashes.

Summary

1. Download and extract the starter files
2. Build and test the application
3. Commit all the existing code without committing any build or output files (or things like `*.swp` or `.DS_STORE` files).
4. Create a `README`.
5. Delete the `Makefile` and modify the `README`, then undo those changes.
6. Create topic branches for the two outstanding issues in the code.
7. In each branch fix the issues.
8. Merge the branches back into `master`.
9. Set up your EECS GitLab account and add your SSH key.
10. Create your EECS GitLab Project/repo for this assignment.
11. Push your commits to this EECS GitLab repo.
12. Create a `report.txt`, commit it, and push the commit.
13. Grade your submission with the autograder.