Homework 5
Git++

EECS 201 Fall 2021

Submission Instructions

This homework will be submitted as a repository on the UMich GitLab server. The repository you submit will be a **private** Project with the name/path `eecs201-basic5` with `brng` added as a Reporter. The branch `dev` will be the one that is checked.

1 Cloning and changing a remote

To get started, create a blank **private** Project without a README under your UMich GitLab account with the name/path `eecs201-basic5` and add `brng` as a Reporter.

Next, clone this repository: `git@gitlab.umich.edu:eecs201/content/hw/eecs201-basic5.git`. We're going to make a local version of every remote branch.

```
# clone the repository
git clone git@gitlab.umich.edu:eecs201/content/hw/eecs201-basic5.git
cd eecs201-basic5

# make a local branch for each remote branch
# remember that you can make for-loops one-liners ;)
for branch in $(git branch -a | grep "remotes/origin" | grep -v "HEAD" | sed -e "s:remotes/origin/::");
do
  git checkout "$branch";
done

# there are other ways to do something similar to this,like forking the repo or cloning it as a mirror
# if you are unable to clone this repo (e.g. it no longer exists),
# you can get the file as a tarball of the repo resulting from the above steps here:
# https://eecs.umich.edu/courses/eecs201/files/assignments/basic5.tar.gz
```

By default when you clone a remote called `origin` will created and its URL will be set to where you cloned it from. We'll need to change the URL of `origin` to point to your project, and then push all the branches to it.

```
# set the url for a remote
git remote set-url origin git@gitlab.umich.edu:<uniqname>/eecs201-basic5.git

# push all branches to the remote
git push -u origin --all
```

This assignment has a decent amount of room for error. You might want to create local "backups" of the branches that you can `git reset --hard` to in case you mess up.

```
# think about what this does and how it works
for branch in $(git branch | cut -c 3-);
do
  git tag "backup-$branch" "$branch";
done
```

This will something known as a "tag" of each branch called `backup-<branch name>`. These tags are named references that point to commits, so you can use them as backup pointers to get back to a particular state with `git reset`. For example, if you've completely messed up `dev`, you can use `git reset --hard backup-dev` to move `HEAD` to where `backup-dev` is (which is where `dev` was at the beginning of the assignment).

There are two feature branches: `bug-space-stdout` and `feat-spongebob`. We'll be looking at both of them in the assignment. In this assignment, the main branch is `dev`. You'll be pushing the **final changes** to `origin/dev`.

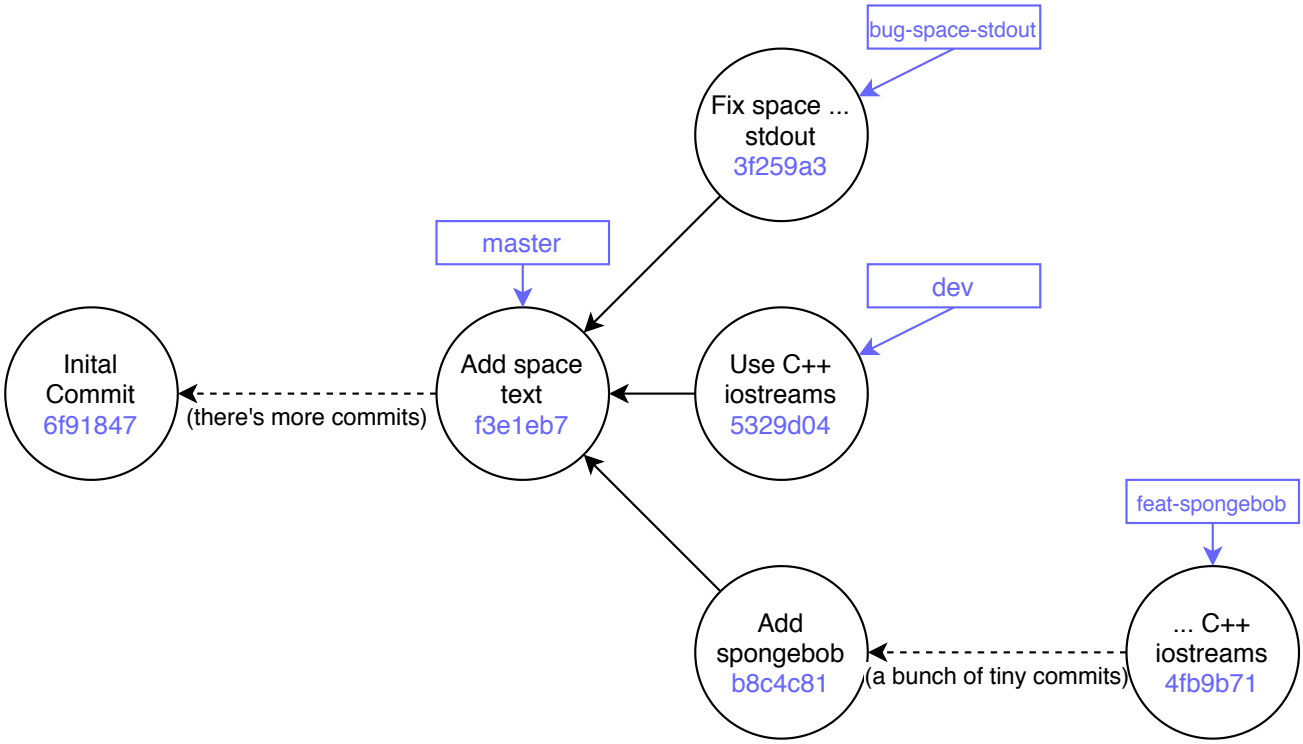

Figure 1: Initial commit graph

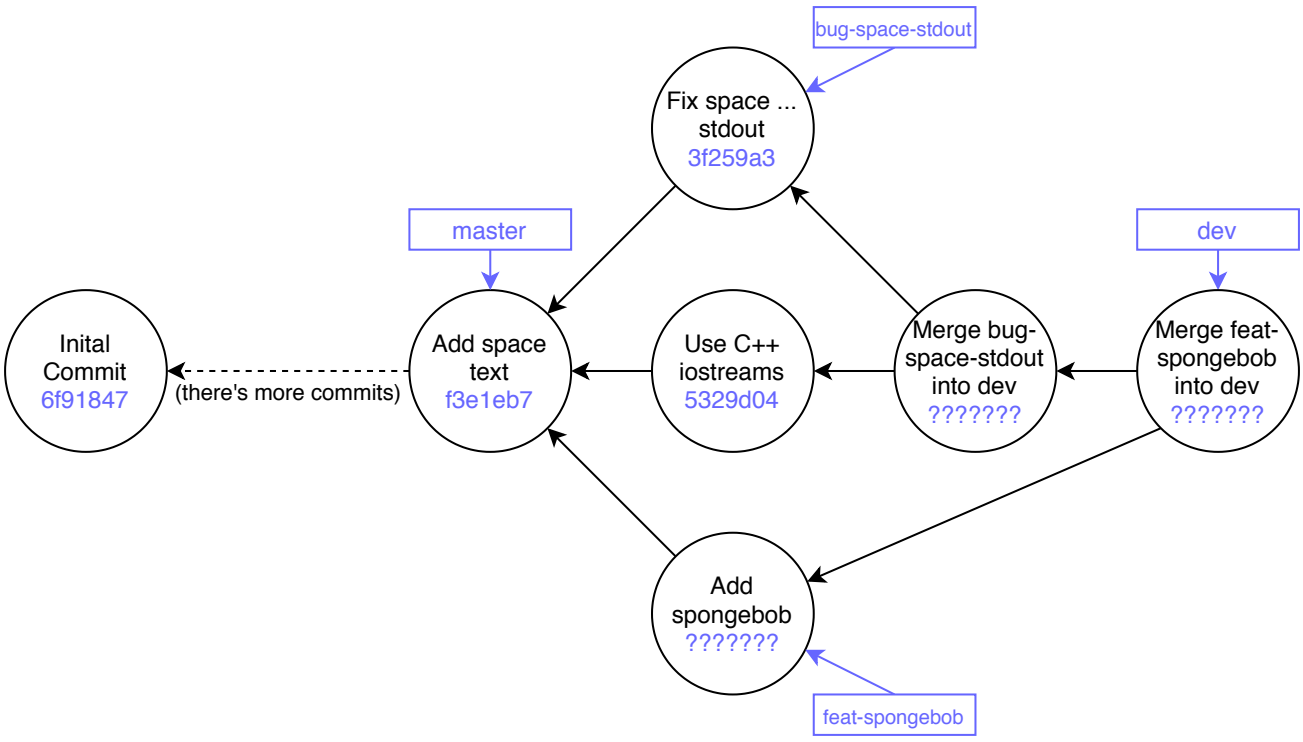

Figure 2: Commit graph after #2 and #3

2 Merge conflict

In this question we'll be merging in a branch that will cause a merge conflict: the `bug-space-stdout` feature branch into `dev`. In this situation, `bug-space-stdout` branched off from `dev` and then `dev` had another commit pushed to it, so a divergence point has formed. What `bug-space-stdout`'s new commit does is fix `space.cpp` so that it outputs to standard output instead of standard error. What `dev`'s new commit does is replace the use of C `stdio` calls (e.g. `printf`) with C++ `iostream` calls (e.g. `cout <<`).

1. Switch to the `dev` local branch.
2. Run `$ make` to build the application that the repo is versioning. If you want to, play around with it.
3. Run `$ git merge bug-space-stdout` to merge `bug-space-stdout` into the current branch (`dev`).
4. This will cause a merge conflict. The merging process will stop for you to deal with the files that have encountered merge conflicts. These files will have parts of them marked off that indicate what the conflicting commits are saying the file should have. These look like `<<<<<<<` and `>>>>>>>`, and are examples of "version control markers".
5. Deal with the files in conflict. If you have trouble understanding what needs to be fixed, investigate the latest commit in `dev` and `bug-space-stdout` to see what their goals are. You can do this by using `git log`, `git show`, and `git diff`.// e.g. `git show 5329d04` will show you what was added/removed in `dev`'s commit. In the end, your fix for the merge conflict should reconcile the ideas in both commits: as a result of this merge, `space.cpp` **should use C++ iostreams to output to standard out (barring the error printout when not enough arguments are provided).** This will involve removing the version control markers and the code reflecting the ideas in the two commits being merged together.
6. Make sure that the code is in compilable condition and that it works as intended.
7. When you are done handling the conflict, stage the appropriate files to mark them as resolved and commit (If you run `git status` it will also communicate this step for you). A merge commit message will automatically be worded for you. This resulting merge commit will contain the changes you made to resolve the conflict.
8. Hooray! You have successfully dealt with a merge conflict!

Some tips:

- The C++ `iostream` equivalent of `stdout` is `cout`
- The C++ `iostream` equivalent of `stderr` is `cerr`

3 Cleaning up with `git rebase`

In this question we'll be using `git rebase` on `feat-spongebob` to clean up its history and get it ready for merging.

1. Switch to the `feat-spongebob` local branch.
2. Run `$ git log`. Note how that there are a bunch of commits that begin with "squashme". Each of these commits contain really tiny changes that don't really warrant commits by themselves for the addition of a new feature. These should be squashed into the "Add spongebob mocking text command" commit.
3. Recall that "squashing" commits puts the contents of multiple commits into one big commit.
4. Run `$ git rebase -i base-tip-commit-hash` to interactively rebase and squash the commits that begin with "squashme" into the "Add spongebob mocking text command" commit. You'll need to figure out which commit to use as the "base tip". Recall that the "base tip" what the rebase happens on top of: commits *after* the "base tip" are what are getting rebased. **The new commit message should discard of all of the "squashme" messages.**
   (In reality, commits that should be squashed might not be nicely labeled "squashme". I personally do it when I know that I'm making a super tiny change that I'll squash in the end, and for this homework it's to more easily communicate to you all what I want to be squashed.)
5. Switch back to the `dev` branch.
6. Run `$ git merge feat-spongebob` to bring that feature back into the `dev` branch.

4 Conclusion

1. Make sure that you are on the `dev` branch.
2. To see that the resulting code compiles, run `$ make`.
3. Try running `$ ./memetext space 'Hello world!'` and `$ ./memetext spongebob 'Hello world!'`
4. Create a file called `report.txt`.
5. On the first line provide an integer time in minutes of how long it took for you to complete this assignment.
6. On the second line and beyond, write down what you learned while doing this assignment. If you already knew how to do all of this, put down "N/A".
7. Add and commit the report file.
8. Run `$ git push` to push the new commits on the local `dev` branch to your campus GitLab project.