Lecture 1 recap

Recap

- Basic commands: `man`, `pwd`, `ls`, `cd`, `mv`, `rm`, `touch`
- Script files
- Running scripts

Some more stuff

- Basic text editor: `nano`
- ^C (ctrl-C): SIGINT
  - "Signal, interrupt"
  - Interrupts running program: can be helpful if the program isn't receiving input
  - More on signals next week...
- Variables
  - Things like `PATH`, `HOME`, `EDITOR`
  - Some programs will look at certain variables
- Argument documentation syntax
  - `command <mandatory arg> [optional arg]`

Announcements

- Basic 1 is out
- Advanced 1 will be out today/tomorrow
- Lecture 1 survey closing at midnight
- When emailing the staff or just me, please include "EECS 201" in the subject line
- We'll be trying out Zoom Meeting this week
    - When I get to the end of a slide, feel free to ask a question

GITing Started

Week 2

```
git init; git status; git log; git add; git commit;
```

Overview

1. What is version control?
2. Basic Git flow
3. Git branches
4. A taste of Git remotes

Version control

- Keep track of changes of files over time, allowing you to roll back to previous versions
- Software to handle this are known as "version control systems" (VCS)

Two paradigms

Centralized (CVCS)

- Central server keeps track of all the changes and history
- Each developer has local copies of files they need, but need to check in with the server to do any versioning
- Sever down? Good luck.
- Examples: CVS, SVN, Perforce

Decentralized (DVCS)

- Each developer has a local copy of the entire codebase and its history
- Developers can perform versioning locally without needing to contact a server
- Server optional
- Examples: Git, Mercurial

Why version control?

- Checkpointing your work
    - Have you ever made `main.c.backup1`, `main.c.backup2`,...?
- Keeping multiple parallel versions of your work
    - Have you implemented {thing} one way, made another implementation of {thing} but wanted to keep both around?
    - Have you ever emailed code or sent code in some messaging app?
    - Have you tried to coordinate people working on the same file?

Enter...

Enter...Git!

- Distributed version control system (DVCS)
- Designed by Linus Torvalds to manage the Linux kernel
- No server needed, super easy to get started with
    - `git init`
    - `git add`
    - `git commit`

That's it, lecture's over!

Git Overview

- Repository: a directory of stuff that Git is versioning
  - `.git` is the directory that holds all this metadata
- Commit: a checkpoint for the files in the repository
  - Given a hash for identification
  - (Unlike other VCS, has actual snapshots of files rather than diffs)
- History is a linked list of commits pointing to their parent
  - Directed acyclic graph (DAG) may be more accurate

Basic commands

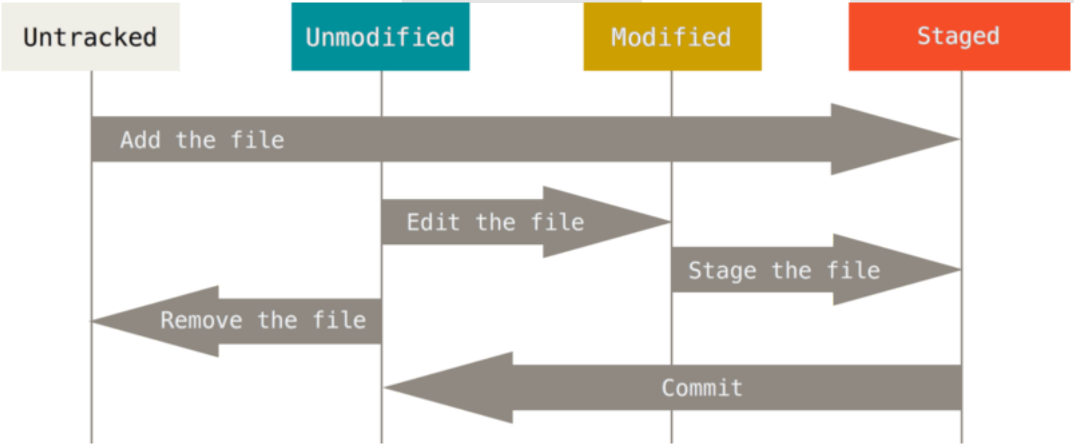
- `git init`
- `git status`
- `git log`
- `git add`
- `git reset`
- `git checkout`
- `git commit`

Some neat resources

- `man git`
- `man git-<command>` or `help git <command>`
- Official Git documentation
- Official Git tutorial
  - `man gittutorial`
- Official Git minimal set of useful commands
  - `man giteveryday`
- *Pro Git* book
  - Free and comprehensive
  - Besides being on the web, has `.pdf`, `.epub`, and `.mobi` formats!
  - A really great read

Files have multiple states

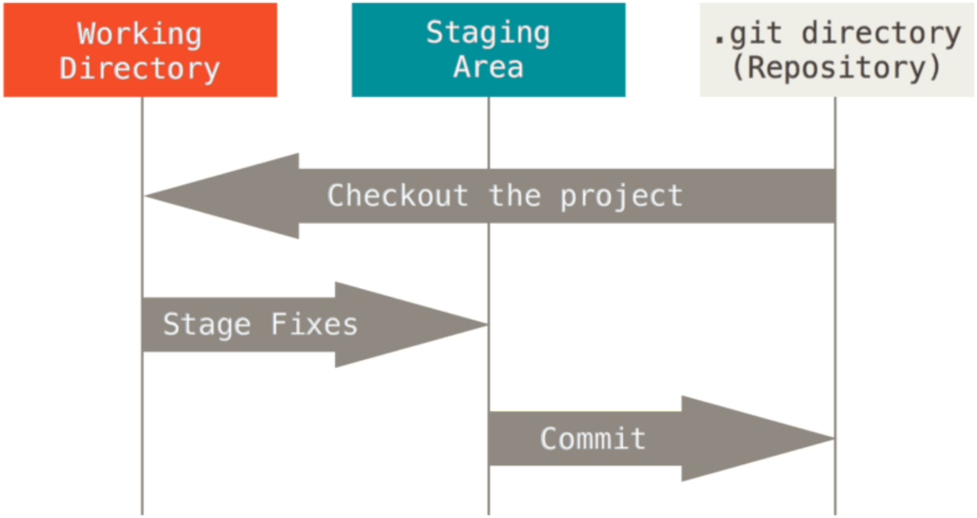
- **Unmodified**: Nothing has happened to this file; no changes compared to current commit
- **Modified**: This file differs from the version in the current commit. Can be `git add`ed to be **Staged**
- **Staged**: This file differs, and is set to be in the next commit
- **Untracked**: This file does not exist in the current commit
  - It's pretty similar to **Modified**; it "differs" by existing while the current commit says it doesn't exist
  - You can hide these from `git status` by adding a `.gitignore` file



From *Pro Git*

Ties into the "areas"

- **Working Directory**: the directory as your filesystem sees it, a mess of files which may or may not be changed, or may be even untracked
- **Staging Area/Index**: list of files whose snapshots will be part of the next commit
  - You'll see it referred to as either: I'm going to say "**index**" for brevity and to distinguish it from the file state of **Staged**
- **Repository**: What commits Git now has saved
- Files and their snapshots will work their way through these three areas



From *Pro Git*

Scenario 1: Untracked file

1. An untracked file chills in the **Working Directory**
2. You decide to start versioning it, so you `git add` it, making it **Staged** and putting it into the **Index**
3. You commit the file in the **Index**, landing it in **Repository**

Scenario 2: Modified file

1. The file is now **Unchanged** as of the current commit, and is still chilling in the **Working Directory**
2. You make some changes, so now the file is **Modified**
    - Oops, maybe I don't like what I did and want to change it back to the old commited version, let's `git checkout` it
3. You `git add` it, making it **Staged** and putting it into the **Index**
    - Oops, maybe I added an extra file I didn't want to stage, let's `git reset` it back to **Modified**
4. You commit the file's snapshot, getting that snapshot into the **Repository**

Putting it together, locally

1. Initialize the repository
    - `git init`
2. Add the initial files you want to track to the **Index**
    - `git add`
3. Commit those initial files to the **Repository**
    - `git commit`
4. Modify some files
    - Don't like a modification and want to make the file **Unmodified** again? `git checkout <filename>`
    - `git restore` is a new command that performs this behavior
5. Add **Modified/Untracked** files to the **Index**
    - `git add`
    - Accidentally added a file? `git reset <filename>` to take it out of the **Index**
6. Commit those files to the **Repository**
    - `git commit`
    - Didn't like your commit message or forgot to include some files? Add them to the **Index**, and `git commit --amend`
7. Goto 4, rinse and repeat

Commits

- `git commit -m <message>` is a quick and dirty way to make a commit
- Not super ideal when it's a project that you're going to collaborate with others on
- `git commit` will open the configured editor and allow you to fully fill out a commit message

Commits

Title

- Limit to 50 characters
- Capitalize the first letter
- Imperative ("Fix xyz", "Remove abc")
- Summarize the commit

Body

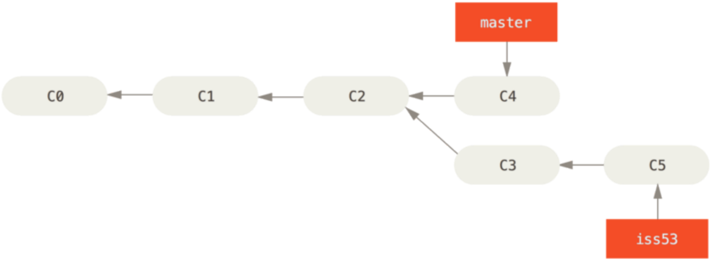
- Limit to 72 characters per line
- Explain what changed and why, not how
  - Your code (ideally) is the "how"
- Depending on your team/workplace: references to bug/issue number e.g. "Issue #22772", "Bug #1337"

No, I'm not making this up, it's straight from the horse's mouth

Ultimately these are just guidelines, not rules. Do what your team does, but try to keep good habits when you start something yourself

Branching

- Making a linked list of commits is cool, but what can we do with it? Can we go back? Can we split off?
- `HEAD` is a pointer pointing to the current commit that's being looked at
- A **branch** in Git is a pointer to a commit
  - Super lightweight compared to other VCS, go wild
  - `HEAD` will follow along with the branch you are on
- Lots of applications:
  - Make a "backup" of branch
  - Manage a feature ("topic"/"feature" branches)
  - Have a separate line of development (e.g. taking two different approaches)
  - Represent release schedules (e.g. a development branch and a release branch)



From *Pro Git*

Branching

- The default branch is `master`
  - Typically used for production/release
- `git branch` lists your local branches
- `git branch <branch-name>` creates a new branch
  - `<branch-name>` will point to wherever `HEAD` is pointing to
- `git checkout <branch-name>` checks out the branch, making your `HEAD` point to where `<branch-name>` is pointing to
  - `git switch` also switches to a branch; added in Git 2.23.0
  - `git checkout -b <branch-name>` creates **and checks out** the branch
- `git merge <branch-name>` will try to move the current branch to where `<branch-name>` is; this is called **fast-forwarding**
  - If the branches diverged (`<branch-name>` and the current branch both got new commits before merging), a special "merge commit" will be produced linking the two branches
  - (This is where things get a bit more messy and complicated: we'll take a closer look later)

Remotes

- So far everything we've been looking at has been local
- What if you want to share it?
- A **remote** is a repository is hosted by some server on the Internet or internal network
- `git clone <URL> [directory]` will copy the repository from the server to your local machine
    - `origin` is the default name of the **remote** whose URL you cloned from
- `git remote -v` will list your **remotes**
- `git fetch` will get the latest commits from the **remote** into the **Repository**
- `git pull` will do a `git fetch` and additionally `git merge`, potentially modifying your **Working Directory**
- As you work on your locally, you can make commits to your local **Repository**
- `git push` will send your commits to the **remote**

Remote hosting services

(a.k.a. Git != GitHub)

- GitHub
- BitBucket
- GitLab
    - GitLab is also a Git host server software that you can use to host your own repos
    - gitlab.umich.edu is the GitLab server that the University of Michigan runs
    - gitlab.eecs.umich.edu is the GitLab server that the EECS runs

Communicating with remotes

- HTTP will use a username and password to authenticate
  - URL format: `https://somedomain.tld/path/to/repo.git`
  - Gets annoying having to type all the time
- SSH requires key setup
  - URL format: `git@somedomain.tld:path/to/repo.git`
  - No need to enter username and password though!
- These are the two most common for day to day use

Questions?

Addenda

Core commands

- `git init`
- `git status`
- `git log`
- `git add`
- `git reset`
- `git commit`
- `git branch`
- `git checkout`
  - (`git switch`)
  - (`git restore`)
- `git merge`

Remote and Collaboration commands

- `git clone`
- `git fetch`
- `git pull`
- `git push`
- `git remote`

Additional Commands

- `git help`
- `git stash`
- `git show`
- `git diff`
- `git rebase`
- `git blame`