Our Goal
- Learn about different kinds of Version Control Systems (VCSs)
- Overview the basics of git
- Touch some intermediate git topics
- Clear up common points of confusion
  - Branch vs. Fork?
  - Merge vs. Pull Request?
  - Pull vs. Fetch?
  - Fork vs. Clone?

What are Version Control Systems
A Version Control System (VCS) records changes to a file set over time, making it easy to review or revert to specific versions later
Motivating example

Why Use Version Control?

- Easy to revert to previous versions
- Work on multiple features in parallel
- Makes collaboration easier
- Narrate the evolution of codebase with messages
- Nice tools such as GitHub (and GitLab (and BitBucket...)) with advanced features such as pipelines, issue tracking, wikis, etc...
- Can store a backup remotely and automatically - easy to keep this up to date!
- Helps keep your working space clean
Who Uses Version Control?
- Programmers
- Applications (Microsoft Word, Google Docs, ...)
- Organizations
  - VCS can be used to sync data, not just code

Two main types of VCS
- Centralized, e.g., SVN
- Distributed, e.g., Git

Types of VCS -- Centralized
- There exists a single "central" copy of the project
  - All developers commit to this single copy
  - Each developer has local working copy(ies)
    - As soon as they commit, the central repo reflects the changes

Centralized version control
- (old model)
- Examples: Concurrent Versions System (CVS)
  - Subversion (SVN)

---

Diagram of Centralized version control
- Server
- Repository
- Working copies
  - Workstation/PC A
  - Workstation/PC B
  - Workstation/PC C
- Alice's working directory
- Bob's working directory
- Yuriy's working directory
- Sarah's working directory
- Fork's working directory
- Server repository

---
Doing work

- I update my checkout (working copy)
- I edit
- I update my checkout again
- I merge changes if necessary
- I commit my changes to the Master

Problems with centralized VC

- What if I don’t have a network connection?
- What if I am implementing a big change?
- What if I want to explore project history later?

Types of VCS — Distributed

- Each developer has their own repository.
  - Created by the developer, or
  - Cloned from an existing (remote) repository
- Developers work on their own repos
  - They can commit, branch, etc.
  - Activity is local unless it is pushed to remote repos
  - Remote activity is not seen until dev fetches from the remote repos
- Examples: Mercurial (hg), git

Distributed version control model
Doing work

- I pull from the Master
- I update my checkout
- I edit
- I commit
- I pull from the Master
- I merge tips if necessary and commit again
- I push my changes to the Master

History view (log)

- Bill and Melinda work at the same time
- At the end, all repositories have the same, rich history

Centralized versus Distributed version control

Centralized version control

Distributed version control

https://github.com/
Our goal with git

Be able to understand the git man-pages

Motivating Example 1: What is this git command?

NAME

git-add - Adds file contents to the index

SYNOPSIS


DESCRIPTION

This command updates the index using the current content found in the working
tree, to prepare the content staged for the next commit. It typically adds the
current content of existing paths as a whole, but with some options it can also
be used to add content with only part of the changes made to the working tree
files applied, or remove paths that do not exist in the working tree anymore.

Motivating Example 2: What is this git command?

NAME

git-checkout - Switch branches or restore working tree files

SYNOPSIS


git checkout [-q] [-f] [-m] [<branch>]

DESCRIPTION

Updates files in the working tree to match the version in the index or the
specified tree. If no paths are given, git checkout will also update HEAD to set
the specified branch as the current branch.
Motivating Example 2: What is this git command?

**NAME**
git-checkout - Switch branches or restore working tree files

**SYNOPSIS**
git checkout [-q] [-f] [-m] [<branch>]

**DESCRIPTION**
Updates files in the working tree to match the version in the index or the specified tree. If no paths are given, git checkout will also update HEAD to set the specified branch as the current branch.

---

**Git Basics -- Tracked vs. Untracked**

- **untracked file**: a file not currently under version control
- **tracked file**: a file that is under version control

---

**Git Basics -- Three Main Stages**

1. **Committed**: Everything in the file is currently in the database
2. **Modified**: Changed the file but have not committed to the database
3. **Staged**: Marked the file for addition to the database in the next commit

Note that all of the above pertain to tracked files.
Git Basics -- Creating Repositories

Initializing a repository

- **git init** - Create an empty git repository or reinitialize an existing one
  - `--bare` - create a bare repository
  - `[directory]` - git init is run inside the provided directory
- **git init** creates a `.git` folder in the directory chosen

Cloning a Repository

- **git clone** - Clone a repository into a new directory
  - `--depth <depth>` - Create a shallow clone with a history truncated to `<depth>` commits
  - `--branch <name>` - Point local HEAD to specific branch (more on HEAD in a bit)
  - `--origin <name>` - Use `<name>` to keep track of remote repo instead of 'origin'
- Basically, clone just:
  - calls init
  - points some meta variables at an existing repository
  - copies the data to the new repo

.git/

- What's in it?
  - `branches/`:
    - COMMIT_EDITMSG: most recent commit message
    - hooks/: contains client or server-side hook scripts (more info)
    - index: The "staging area"
    - info/exclude: global exclude file for your project
    - refs/: keeps track of history of HEAD and refs
    - objects/: where the actual content is stored
    - tags/: keeps track of tags and refs

.git/

- What's in it?
  - `branches/`:
    - COMMIT_EDITMSG: most recent commit message
    - hooks/: contains client or server-side hook scripts (more info)
    - index: The "staging area"
    - info/exclude: global exclude file for your project
    - logs/: keeps track of history of HEAD and refs
    - objects/: where the actual content is stored in a database
    - refs/tags/: keeps track of tags and refs
Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream

Git Vocabulary

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree
- staged
- commit
- ref
- branch
- HEAD
- upstream
Git Vocabulary

- **index**: staging area (located .git/index)
- **content**: git tracks what's in a file, not the file itself
- **tree**: git's representation of a file system.
- **working tree**: Tree representing what is currently checked out (what you see)
  - staged
  - commit
  - ref
  - branch
  - HEAD
  - upstream

- **staged**: ready to be committed (in index/will be stored in a commit object)

- **commit**: A set of database entries detailing a snapshot of the working tree
  - ref
  - branch
  - HEAD
  - upstream

- **ref**: pointer to a commit object
Git Vocabulary

- **index**: staging area (located .git/index)
- **content**: git tracks what's in a file, not the file itself
- **tree**: git's representation of a file system.
- **working tree**: Tree representing what is currently checked out (what you see)
- **staged**: ready to be committed (in index/will be stored in a commit object)
- **commit**: A set of database entries detailing a snapshot of the working tree
- **ref**: pointer to a commit object
- **branch**: basically just a (special) ref. Semantically: represents a line of dev
  - **HEAD**
  - **upstream**

Git Basics

**Working Locally**

- index: staging area (located .git/index)
- content: git tracks what's in a file, not the file itself
- tree: git's representation of a file system.
- working tree: Tree representing what is currently checked out (what you see)
- staged: ready to be committed (in index/will be stored in a commit object)
- commit: A set of database entries detailing a snapshot of the working tree
- ref: pointer to a commit object
- branch: basically just a (special) ref. Semantically: represents a line of dev
- **HEAD**: a ref pointing to branch/commit being worked on (i.e., Working Tree)
- **upstream**
Git Basics: Changing Content -- `git add`

`git add` does two things:

1. given an untracked file it will
   a. start tracking it
   b. update `.git/index` using the current content found in the working tree to prep the content for the next commit (i.e., the content is staged)
2. given a modified unstaged file it will
   a. stage its contents for commit

`--patch, -p` start an interactive staging session that lets you choose portions of a file to add to the next commit.

Git Basics: Changing Content -- `git commit`

`git commit` updates the Git database with staged content in `.git/index`

- Note that staged files can have unstaged changes
- By default this will open an editor for you to enter a commit message

`--message=<msg>, -m <msg>` Add `<msg>` as the commit message. If multiple messages are given, concatenate as separate paragraphs

`--patch, -p` Use the interactive patch selection interface to choose which changes to commit (similar to `git add -p`)
Git Basics: Making Queries

---

Git Basics: Making Queries — git status

`git status` shows the working tree status. This command displays:

- paths that have differences between the index file and the current HEAD
- paths that have differences between the working tree and the index file
- paths in the working tree that are not tracked by git

`--short`, `-s`: Give the output in the short-format
`--ignored`: Show ignored files

---

Git Basics: Making Queries — git log

`git log` inspects commit history with multiple display options

- `git log` is basically a wrapper around `git rev-list` and `git diff-*` (don’t worry about these - I sure don’t!)

Some Examples

```
git log
```

```
git log --graph
```

```
git log --graph --all
```

```
git log --graph --all --oneline
```

---

Git Basics: Making Queries — git log

...Some Examples

```
git log --graph --abbrev-commit --decorate --format=format:'%C(bold blue)%h%C(reset) - %C(bold cyan)%ad%C(reset) %C(bold green)(%ar)%C(reset) %C(bold cyan)(committed: %cD)%C(reset) %C(auto)%d%C(reset)%n''
%C(white)%s%C(reset)%n''
%C(dim white)- %an <%aE>
%C(reset) %C(dim white)(committer: %cn <%cE>)%C(reset)'
```

---
Git Merge

NAME

git-merge - Join two or more development histories together

SYNOPSIS

   [-v] [git-strategy-option] [-f] [-k]
   [-f] [-r] [-f] [-s] [commit]... [commit]...

DESCRIPTION

`git merge` combines changes from the named commits (along the time their histories diverged) into the current branch. This command is used by `git pull` to incorporate changes from another repository and can be used by hand to merge changes from one branch into another.

Assume the following history exists and the current branch is “master”:

```
A --- B --- C
     |    |
     v    v
D --- E --- F
```

The command `git merge E` will result in:

```
A --- B --- C
     |    |
     v    v
D --- E --- F
     |    |
     v    v
B --- C --- F
```
Git Rebase

Changing Commit History with Rebase

- Git rebase lets us change our commit history
- rebase is a powerful tool, but we will only scratch the surface
Changing Commit History with Rebase

- Git rebase --onto gives us a bit more power
Why use Git Rebase?

- Clean history
- If you are issuing a pull request it's nicer for folks looking over it to decide to include it
- Clean "narrative"
- Branches can look messy

Points of Confusion

Fork vs. Clone
Fork vs. Clone

Fork

- Fork is NOT A GIT CONCEPT
- It was invented by GitHub
- Fork stores extra information and makes pull requests possible

Clone

- Clone IS A GIT CONCEPT
- Clone extends init
- Clone exists independent of github

Branch vs. Clone

Branch

Branch creates a ref

Clone

Clone creates a new repository

Pull vs. Fetch
Pull vs. Fetch

Fetch
- Take target branch from a remote repository and store it in .git/refs/remotes/
- NOT integrated/merged with local branch/repository

Pull
- Fetches remote branch and merges with local branch or repository

Next class: Thursday (September 17)
- First in-class exercise
- On using git (today is a prelude with useful info)
- Form 4- or 5-person teams
  - Use Moodle to self-select a team; can do it now until Friday at noon
  - At least one person per team needs to bring a laptop

Homework 1
- Three in a Row game
  - Implemented in Java using Swing
- Topics:
  - Code review
  - Architecture patterns
  - Design patterns
- Due: September 28, 2020 9 AM EDT
  
  https://github.com/LASER-UMASS/cs520-Spring2020