# CS 45, Lecture 10

Version Control II

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

**Outline**

## Contents



## 1 Review

### Overview

Last lecture, we saw:

- How to keep track of linear (ordered) histories of files
- How to turn non-linear version history into pseudo-linear history

In this lecture, we will see:

- How to resolve merge (or rebase) conflicts
- How to collaborate on files with others over the internet
- How to back up your files and their history on the internet

### Git is Confusing

- Git is confusing!
- Ask as many questions as you need, and don't let me move on if you don't yet understand something.

```
commit c8f3dcfb80e87d4aa334f6bcdd541ddb78135881 (origin/master, origin/HEAD)
Merge: 480cbe2 2a7f97b
Author: engler <ddd.rrr.eee@gmail.com>
Date:   Thu Feb 13 14:27:50 2020 -0800

    git.  sucks.

    Merge branch 'master' of github.com:dddrrreee/cs140e-20win
```

### Terminology

**HEAD** is the branch you're currently looking at

**a branch** is a named version of a repository

**fast-forwarding** means moving an old branch "forward" to add new commits from a more recent branch

**merging** is a way of combining branches by creating a single "merge commit"

**cherry-picking** is a way of moving commits from one branch to another

**rebasing** is a way of moving an *entire branch* to have a different "base"

## 2 Merge Conflicts

### Merge Conflicts

**Definition 2.1** (merge conflict)**.** A merge conflict is what happens when you try to combine two contradictory branches. Git can't always figure out how to resolve the contradiction, so it'll ask the user (you).

- Git normally resolves merge conflicts automatically.
- Some conflicts have multiple valid resolutions (e.g., what if one person edited a file that another person deleted?).
- If Git doesn't know what to do, it'll ask you to resolve the conflict.

### Merge Conflicts

Git will tell you which files conflicted, and tell you to resolve the commits and commit the results:

```
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

### Merge Conflicts

*Conflict Markers*

Git will also add conflict markers to the files:

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=======
```

```
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

This might look scary, but it's not that bad!

### Merge Conflicts

*Conflict Markers: The Base Branch*

The top part (labeled `HEAD`) are the changes in the base branch (the branch you're currently on):

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=======
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

### Merge Conflicts

*Conflict Markers: The Incoming Branch*

The top part (labeled with a branch name or commit message) are the changes in the incoming branch (the one you're merging):

```
Hello, my name is Akshay Srivatsan.
<<<<<<< HEAD
I'm doing my PhD in the Stanford CS department.
=======
I am a PhD student studying CS at Stanford.
>>>>>>> add-major
I'm currently co-teaching CS45 and doing research.
```

When you see these conflict markers, all you have to do is make the files look the way you want them to look at the end. In this case, I added the text "I'm doing my PhD in the Stanford CS department." on `main`, but I added the text "I am a PhD student studying CS at Stanford." on the branch `add-major`. When I tried to merge `add-major` into `main`, Git didn't know what to do, so it's asking me. Now I can choose either of the two sentences to keep, and delete the other (or I could keep both of them, if I wanted).

As an aside, you might see the name `HEAD` pop up in Git. This basically just means "what commit you're currently looking at".

### Merge Conflicts

*Resolving a Conflict*

Pick how you want to resolve the conflict (i.e., decide what the "correct" result of the merge is), and make the file look that way!

```
Hello, my name is Akshay Srivatsan.
I'm a PhD student in the Stanford CS department.
I'm currently co-teaching CS45 and doing research.
```

In this case, I mixed together both versions. The "correct" answer often depends on what exactly you're doing, which is why Git can't figure it out for you.

### Merge Conflicts

*Commiting the Merge*

Resolve all the conflicts in all the files however you want, then:

1. `git add` your changes to track them
2. `git commit` the changes (with no message)

Git will auto-generate a message, and open your `$EDITOR` to have you confirm it:

```
Merge branch 'add-major'
```

Save the file in your editor and close it (`:wq` in Vim), and Git will save the merge commit. That's it—the merge conflict is gone!

That's all you have to do—make the files look "correct", then commit! A merge conflict really isn't as bad as people sometimes make it sound; all it means is that there are multiple ways to merge the two branches, and Git wants you to pick one.

If you decide to use `rebase` instead, the process is pretty much the same—just run `git rebase --continue` instead of `git commit` at the end.

### Merge Conflicts

*Rebase Conflicts*

Resolve all the conflicts in all the files however you want, then:

1. `git add` your changes to tell Git you fixed them
2. `git rebase --continue`

Since rebasing doesn't create a merge commit, you don't run `git commit`; use `git rebase --continue` instead!

Remember, rebasing happens *backwards*; the base branch (the one onto which you're rebasing) becomes `HEAD`, and the "feature" branch becomes the incoming branch.

### Resolving Merge Conflicts

To resolve a merge conflict:

1. Don't panic!
2. Look at the files in conflict (run `git status` to see what's going on).
3. Fix each conflict, one-by-one.
4. When you're done, `git add` all the fixed files and `git commit`.

Let's practice!

Merge conflicts usually happen in shared repos, so let's CLONE one of my repos onto your computer:

```
git clone https://github.com/Akshay-Srivatsan/cs45-23win-demo-repo.git
```

We'll go into more detail about how shared repositories work in the last section of this lecture, but for now:

- You can "clone" a shared repository using `git clone`, which makes a local copy.
- You can "fetch" commits from the shared repository into yours using `git fetch`. The commits will go into a separate branch so they don't conflict with yours; by convention, the branch names have the prefix "origin/" prepended to them, so `main` goes into `origin/main`.

#### Pulling Changes

You might have seen references to the `git pull` command before. This is a combination of two commands, but the exact two depends on your Git version and configuration:

**`git pull --ff-only`:** `git fetch` and `git merge --ff-only` (Default)

**`git pull --no-rebase`:** `git fetch` and `git merge` (Old Default)

**`git pull --rebase`** `git fetch` and `git rebase`

Depending on your preferences, you can configure `git pull` to do any of these.

I personally use `git pull --rebase` the most often, since I don't like having merge commits in my repo history.

## 3 Commit Etiquette

#### Commit Messages

Git only saves work that we've committed, so we want to commit as often as possible, but…

- Other people will also look at your commit history to see what you did.
- Your commit messages in the history should be short and specific, but descriptive enough that someone new can understand what they do.
- Similarly, each of your commits should do a single thing, so a single message can describe it easily.
- Good commits are BISECTABLE; you should be able to checkout any commit in `main` and get a valid (e.g., compilable) state of your repo.

Writing good commit messages is part of being a good programmer!

These goals might seem contradictory; how do we commit as-often-as-possible, but still make sure each of our commits are meaningful and discrete? The answer is: we don't! While we're developing, we commit as often as we want. Then, when we're ready to share our work with others, we *edit our commit history* to make it look like we made nice, easy-to-understand, discrete commits.

#### Squashing Commits

We can commit often locally but still have meaningful commits in the end by SQUASHING commits together with INTERACTIVE REBASE.

##### *Editing History*

Interactive rebasing edits history! Don't do this on a branch you share with other people (like `main`). In general, only do this on commits you **have not** pushed. Otherwise, you'll have to FORCE-PUSH (`git push --force`) your changes, which will **destroy** everyone else's changes.

You can start an interactive rebase using the command `git rebase --interactive <base>`; for example, `git rebase --interactive main` will let you edit every commit that's in your branch but not in `main`.

### Interactive Rebasing

Git will open `$EDITOR` with a list of actions (which you can edit!).

```
pick 0cd3296 start working on new file
pick 594a80c continue working
pick 162392b almost done
pick bf45520 done
pick c545ae9 oops, had a bug
pick 9b3d056 fix the bug for real this time
```

Each line represents one commit. The first word is a "command"; `pick` cherry-picks (i.e., includes) the commit in the new history, `reword` lets you edit the commit message, `edit` lets you change the commit contents, `squash` and `fixup` both squash the commit into the previous one, and `drop` removes the commit.

You might notice that the default behavior here is to cherry-pick every commit. This is the exact same as a normal (non-interactive) rebase! What we called "rebasing" earlier is actually a special case of editing history, but you can go far beyond that with interactive rebasing.

### Squash and Fixup Commits

Squash commits let you specify that two commits are closely related, so they should be combined into a single commit with both messages.

Fixup commits let you specify that a particular commit just "fixes" a previous one, and therefore should be absorbed into the previous commit.

```
reword 0cd3296 start working on new file
squash 594a80c continue working
squash 162392b almost done
squash bf45520 done
squash c545ae9 oops, had a bug
squash 9b3d056 fix the bug for real this time
```

Note that we could also use `fixup` here, the only difference is whether the original message gets saved or thrown away. In this case, we're using `reword` on the first commit anyway, so it's a moot point.

### Rewording Commits

When you want to `reword` a commit, Git will open `$EDITOR` and ask you for a new commit message. Enter the message you want, save, and quit.

```
Add a file providing more information about the project

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Fri Feb 3 22:34:02 2023 -0800
```

### Amending Commits

If you want to edit the most recent commit you made (i.e., `HEAD`), you can skip the rebase and just AMEND it, using `git commit --amend`.

This will also let you edit the commit message of the last commit.

If you want to add to an earlier commit but don't want to do the full interactive rebase yet, you can use `git commit --fixup <hash>` to mark a commit as being a fixup commit of an earlier commit.

You can then use `git rebase --interactive --autosquash <base>` to automatically absorb your fixup commits into the original commits.

Again, only do this if **no one else** is using your branch.

### Good Commit Messages

From Stanford Stagecast: Proleptic:

```
b68f706 Add training trick for handling missing notes
7c0afae Fix audio issue in metronome
fa18a3d Add a metronome for beat tracking
739aa0a Add test for per-millisecond prediction on MIDI files
dc4693a Use last eight piano roll columns to predict next
eb327b4 Fix timing bug in MIDI file parser
b46c6f6 Add MIDI training demo program
a624b32 Switch to cross-entropy loss for MIDI classifier
db3c6f3 Fix bug in MIDI parser
5b0f299 Return effective learning rate from training wrapper
726886d Print more relevant training info from full-piano predictor
```

These commit messages aren't perfect, but they're short, descriptive, and make it clear what one thing each commit does. They generally follow the format VERB-OBJECT (although the implied subject isn't always consistent), and they describe which part of the codebase they're touching.

These commits are also bisectable; that means, if I notice a bug, I can *binary search* to figure out which commit introduced the bug. Git actually has a tool for this built-in, called `git bisect`—you give it a start and end commit (the start commit definitely doesn't have the bug, and the end commit definitely does), and it'll checkout commits in between to help you figure out where the bug was introduced.

### Bad Commit Messages

From CS 140E, Winter 2023:

```
a527839 minor
adaf72e minor
c9c6193 minor
d64a6ef minor
ff2636e minor
4a988f2 minor
cb901d5 minor
8d4e80a minor
53b5e84 minor
0321f79 minor
4126899 minor
f1d7231 minor
cefba82 minor
```

```
44e7773 minor
571c20b minor
059cb3f minor
eaa75ae minor
ebbe9db minor
13570e0 minor
3e51470 minor
95a0fad minor
5d2c780 minor
d5caf55 minor
c26b868 minor
080ddf2 minor
f492a3f minor
```

```
2e67fd8 minor
e530f6e minor
70387f2 minor
e3d971e minor
91b236e minor
de176a8 minor
461e76a minor
48cd0ff minor
0543316 minor
40b48f6 minor
fb0ec84 minor
3a124af added basic files.
```

These commit messages are useless; if you try to look back at your commit history, you're going to have no idea what's going on. If one of these commits had a bug, it's hopeless to try and figure out which one introduced it. Additionally, messages like this are often a symptom of poorly-separated commits; it's

impossible to describe what a commit does because each commit does way too many things (or, alternatively, a single discrete change is scattered across many commits, so there's no meaningful description of what a single commit did).

For the record, these are both real sequences of commit messages from projects I've worked on. You'll probably run into both ends of this as you work with different groups of people, but whenever you can, try to make your commit messages more like the first example.

# 4 GitHub

### GitHub

- One of the most common reasons to use Git is to be able to collaborate.
- Git has built-in support for REMOTE repos, which exist on the internet somewhere.
- You CLONE a remote repo to get a local copy. You can then make commits on the local repo. The remote repo is conventionally named `origin`.
- You FETCH while inside a clone, which copies the remote `main` branch into a branch called `origin/main`.
- You MERGE or REBASE your local `main` into/onto `origin/main`.
- You PUSH your new `main` back to the remote, which updates its `main` and your `origin/main`.

Remember, you can combine `fetch` and `merge` (or `rebase`) using `pull`, if you want to.

You can actually have multiple remote repos for a single local repo. For example, you might have `origin` as your copy of the repo on GitHub, and `upstream` as someone else's copy of the same repo from which you want to cherry-pick changes.

### GitHub Demo

Let's create a new repository on GitHub!

You'll need the `git` command and the GitHub CLI (`gh`).

1. Go to `https://github.com/new` and pick a name.
2. Click "Create repository" to continue.
3. Run `git clone` with the URL of your new repo.
4. Run `gh auth login` from inside your new clone. Tell `gh` that you want to use it to authenticate with `git`.
5. Make some changes (add a file), and run `git push` to upload them!

### GitHub Demo

Let's start collaborating!

1. On the GitHub website for your repo, go to "Settings" and click on "Collaborators".
2. Add the person sitting next to you as a collaborator!
3. Make a clone of their repo, make some changes, then commit and push them. Use `git fetch` or `git pull` to download their changes to your repo.
4. What happens if you both try to edit the same file at the same time?

5. Can you push a new branch to your partner's repo?[1]

### Pull Requests

- It's dangerous to give access to the `main` branch on your repo to everyone; someone might start messing with it!
- In "Settings/Branches", you can enable BRANCH PROTECTION for `main`. Specifically, you can enable "Require a pull request before merging".
- A PULL REQUEST[2] is a way to review a change before merging it. You (the repo owner/maintainer) can choose whether to approve or reject the request.
- To create a pull request: create a new branch, make your changes, push your new branch, then run `gh pr create`.

### When to use Git

- When you want to look at past versions of a folder.
- When you want to be safe from accidentally overwriting your work.
- When you want to collaborate with other people asynchronously (use GitHub!).
- When you want to keep a backup copy of a folder with full history (use GitHub!).
- You want to "fork" a project already using Git/GitHub and contribute back to it.

I personally use Git pretty much whenever I write code, and even sometimes when I'm writing prose. Even my lecture slides for this class are tracked in Git… and one of the lectures in Winter Quarter only happened because Git saved my slides from accidental deletion.

I actually use Git so often that I have a bunch of aliases, both in my shell and in Git itself, to make using it faster. From my `.zshrc`:

```
alias ga="git add"
alias gc="git commit"
alias gc!="git commit --amend"
alias gcmsg="git commit --message"
alias gca="git commit --all"
alias gcam="git commit --all --message"
alias gca!="git commit --all --amend"
alias gp="git push"
alias gf="git fetch"
alias gfm="git pull --no-rebase"
alias gfr="git pull --rebase"
alias gff="git merge --ff-only"
alias gst="git status"
alias gr="git rebase"
alias gri="git rebase --interactive"
alias glog="git log"
alias gco="git checkout"
alias gb="git branch"
```

And from my `.gitconfig`:

[1] Hint: you might have to use the `--set-upstream` flag; Git will tell you exactly what to do.
[2] This is misleadingly named, it's really a "merge request"

```
[alias]
  co = checkout
  c = commit
  st = status
  b = branch
  hist = log --pretty=format:"%Cred%h%x09%Cgreen%cs%x09%Creset%s%x20%Cblue[%an]%Creset"
  uncommit = reset --soft HEAD^
  amend = commit --amend
  histedit = rebase -i origin/main
  unstash = stash pop
  unadd = restore --staged
  skip = update-index --skip-worktree
  unskip = update-index --no-skip-worktree
  skipped = ! git ls-files -v | grep '^S' | cut -d' ' -f2
  list = ls-files -v
  ff = merge --ff-only
  delete-remote-branch = push origin --delete
```

I wouldn't recommend copying all of these, since some of them are particular to the way I use git, but they might give you ideas of ways you can make your own Git usage more convenient. A git alias can be run as a git subcommand, so I can run `git unadd hello.txt` instead of `git restore --staged hello.txt`.