Version Control (Lecture 1)

Peter Ganong, Maggie Shi, and Will Pennington

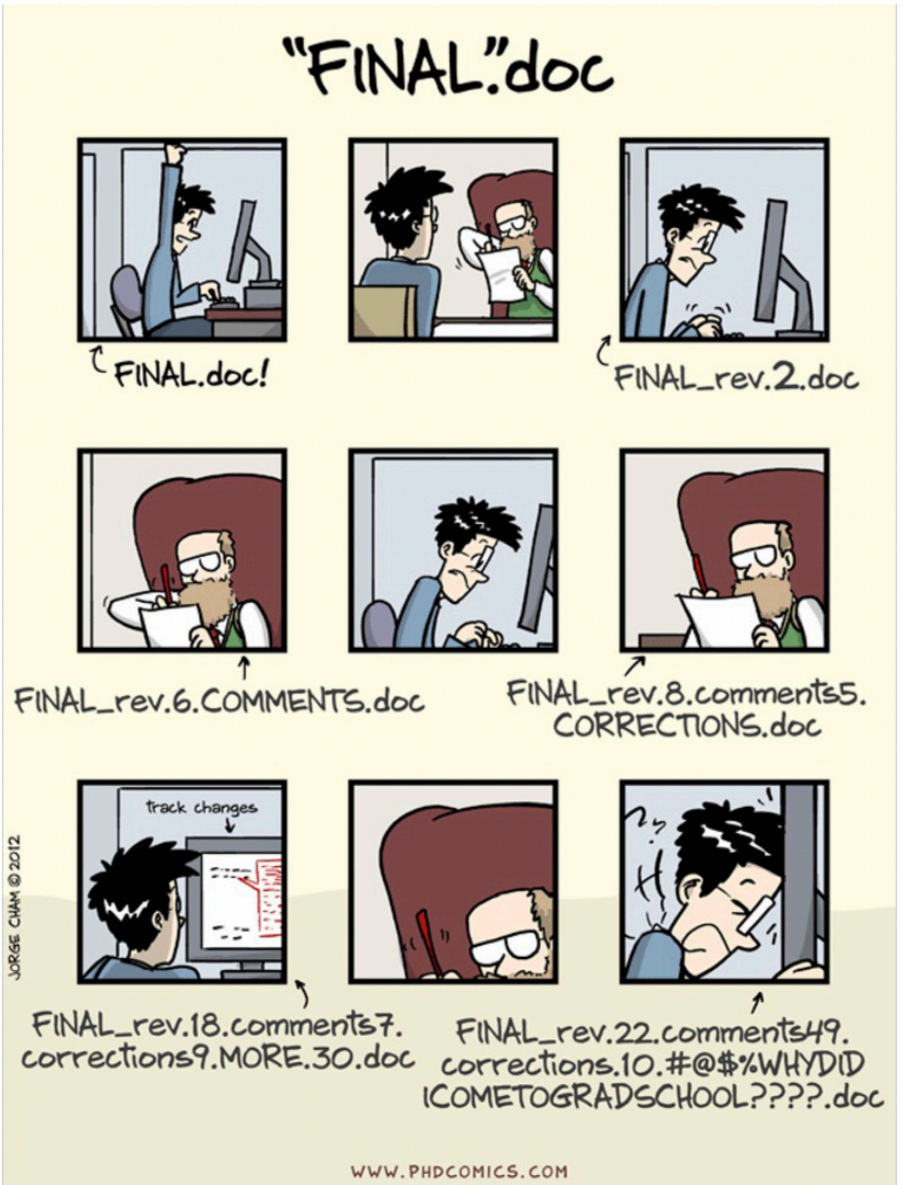
2025-12-31

Conceptual

Roadmap

- What is version control?
- What is Git and why do I need it?
- How (not) to learn git

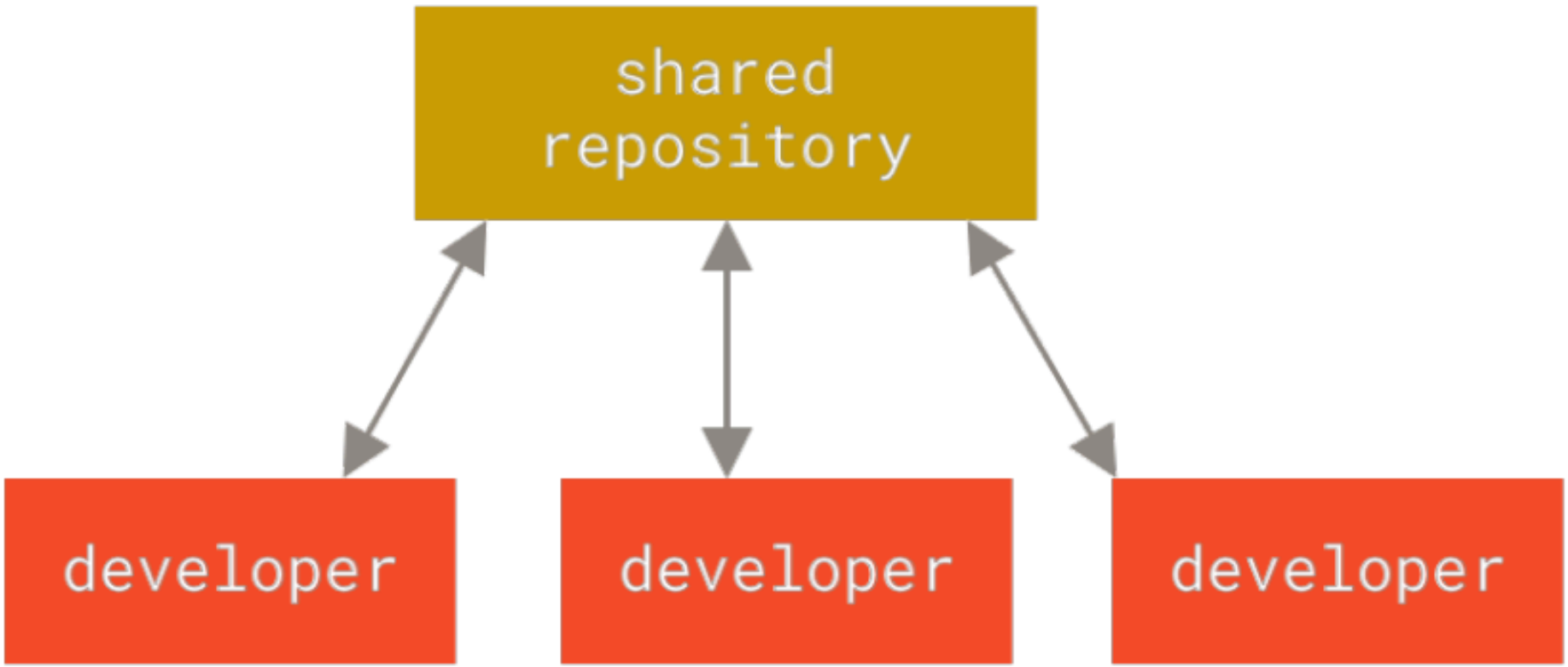
What is version control?



What is version control?

- Version control: a system that records changes to a file or set of files over time so that you can recall specific versions later
- Examples of version control
  - Informal: date multiple versions of a document
  - Tools like dropbox and google docs do this automatically

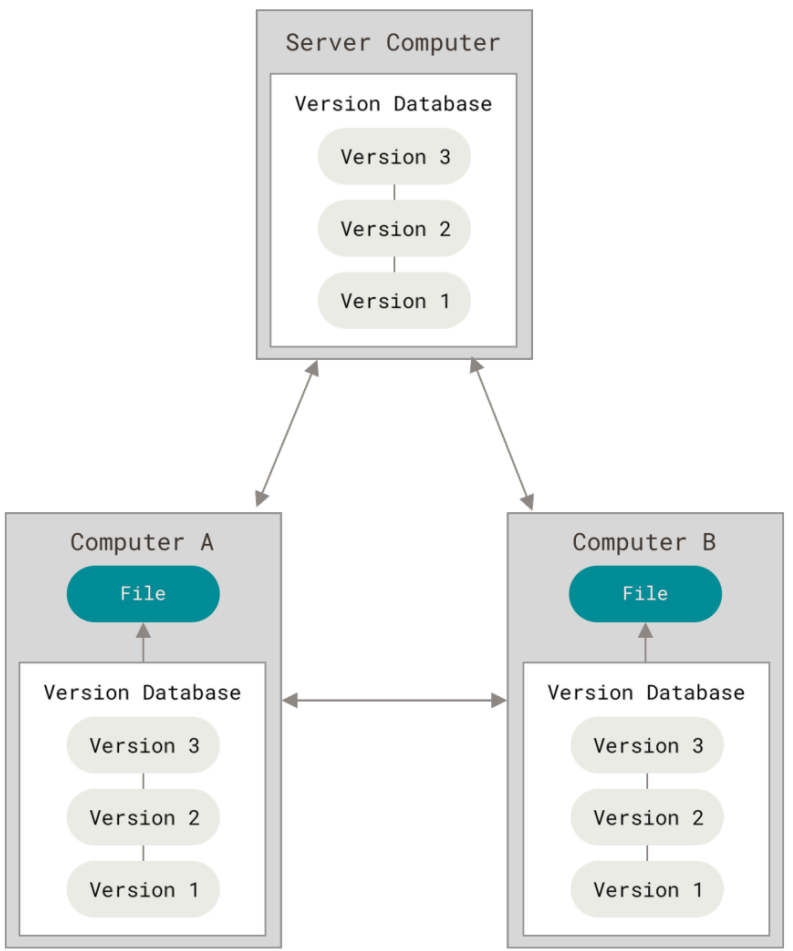
What is centralized version control?



Note: a repository is basically a folder directory with extra capabilities

Git: “distributed” version control

Each computer fully mirrors the Git repository, including its full history



Define terms

- Git: software used for version control locally (i.e., on your computer)
- Github: hosting service for git repositories – i.e., a place to store them online

Why should you learn Git in general?

- Git never loses anything.
- Useful for solo projects for version control (as opposed to: `final_report_revised_v5_maggieedits.qmd`)
- Crucial in group projects to make sure you don't write over others' work or break something.
- Crucial for "teamwork" with AI agents (typical workflow is that AI agent proposes work for you to review)

A skeptic's question

> If dropbox and google docs can track changes automatically (and there are similar tools for code like Google Colab and VSCode for the web) then why do we even need to use git?

Answer

How will you use git in this class?

- Download lecture notes and assignments
- Submit psets and final project
- Collaboration on psets and final project – can work parallel as opposed to sequentially, and work without fear of writing over others' work

How to learn git

1. Accept that this is tricky and you will make mistakes
2. We are going to try a TON of different ways to teach this
    - graphical
    - code examples
    - do-pair-share in class
    - video game on pset
    - exercises on pset

How not to learn git

Summary

- Version control lets you experiment freely
- We will teach git in lots of different ways
- Understanding git is crucial for collaborating with humans or AI agents

Roadmap to the rest of lecture

Sources: Textbook: Pro Git, we will only cover a short bit; Video: Git for ages 4 and up

Section	Pro Git Chapter	Uses internet?
~~Conceptual~~	Chapter 1 (Getting Started)	–
Track One Version on Local	Chapter 2.2 (Git Basics)	No
Branching	Chapter 3.2 (Git Branching)	No
Merging	Chapter 3.2 (Git Branching)	No
Merge Conflicts	Chapter 3.2 (Git Branching)	No
Reconciling with Remote	Ch 2.5 and 3.5	Yes[1]

Command line + Track One Version on Local (Chapter 2.2)

Roadmap

- Command line
- Git workflow
- File lifecycle
- Basic Commands

Why use the command line

- Direct control over files, processes, and version control tools like Git
- Automates repetitive tasks otherwise done via point-and-click in an app
- Same commands across environments—great for collaboration and reproducible workflows
- This is how AI agents work. To understand what they are doing for you you need the command line.

Review of commands

Action	macOS (Terminal) or Windows (PowerShell)
Show current folder	`pwd`
List files in folder	`ls`
Navigate inside a folder	`cd <folder>`
Navigate up one level	`cd ..`
Copy a file	`cp <source_file> <new_file_name>`

Action	macOS (Terminal)	Windows (PowerShell)
Navigate to root folder	`cd ~`	`cd \`
Make an empty file	`touch <filename>`	`New-Item <filename>`
View a text file quickly	`cat <filename>`	`Get-Content <filename>`

Create a new repo

We will use `%` to indicate material run at command line. On some computers it will appear as `PS C:\>`.

1. `% cd <directory_for_repo>` – where we want our new repo to be inside of
2. `% git init <repo_name>` – creates a new repo called `<repo_name>` that has git capabilities
3. `% cd <repo_name>` to navigate to inside the new repo

File lifecycle – statuses

- Untracked – file is not tracked by git in any way
- Version controlled files
  - Unmodified – file has not changed since last “commit” (the last “version” of this file tracked by git)
  - File has changed since last commit
    - Modified – user has not yet asked git to record the change for next commit
    - Staged – user will soon ask git to record the change for next commit

the three most important git commands

command	what it does
`git status`	check which files have changed
`git add <filename>`	stage a file
`git commit -m "<message>"`	record changes to all staged files

We will now walk through a first worked example of how to use git

worked example (1 of 4)

Create a new repo called `test`

```
% git init test
% cd test
% ls
<empty>
```

I make and save a new file called foo.txt

```
% echo "hello world" > foo.txt
```

I can see this file in my directory.

```
% ls
foo.txt
```

And I can print the contents of this file

```
% cat foo.txt
hello world
```

Interpretation of step 1


```
% git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    foo.txt
```

We created an “untracked” file

worked example (2 of 4)

I ask git to record my change (“stage”)

```
% git add foo.txt
% git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   foo.txt
```

Interpretation of step 2

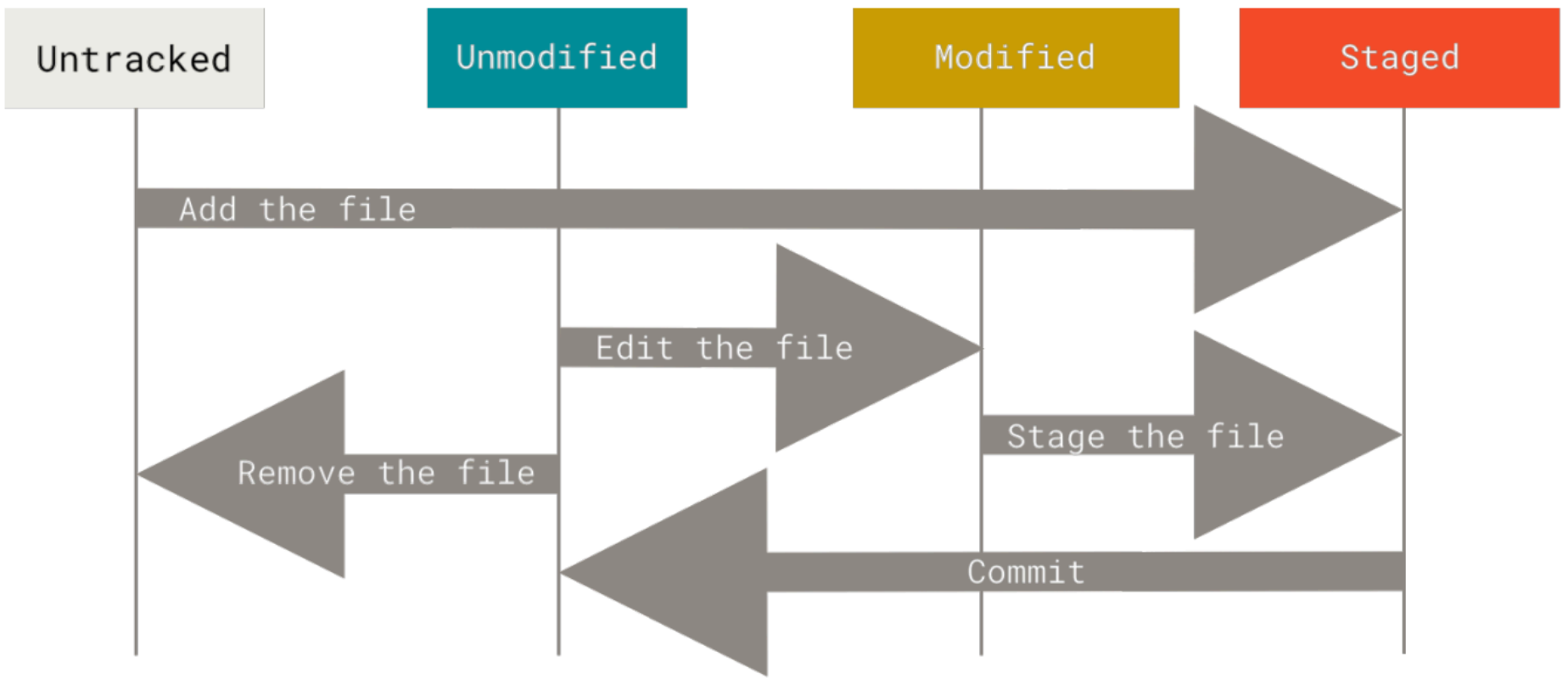


Figure 8. The lifecycle of the status of your files

What did we do? we moved the file the full distance across the board, from “untracked” to “staged”.

worked example (3 of 4)

```
% git commit -m "create foo.txt"
[master (root-commit) a6e1b8a] create foo.txt
 1 file changed, 1 insertion(+)
 create mode 100644 foo.txt
% git status
On branch master
nothing to commit, working tree clean
```

Interpretation of step 3

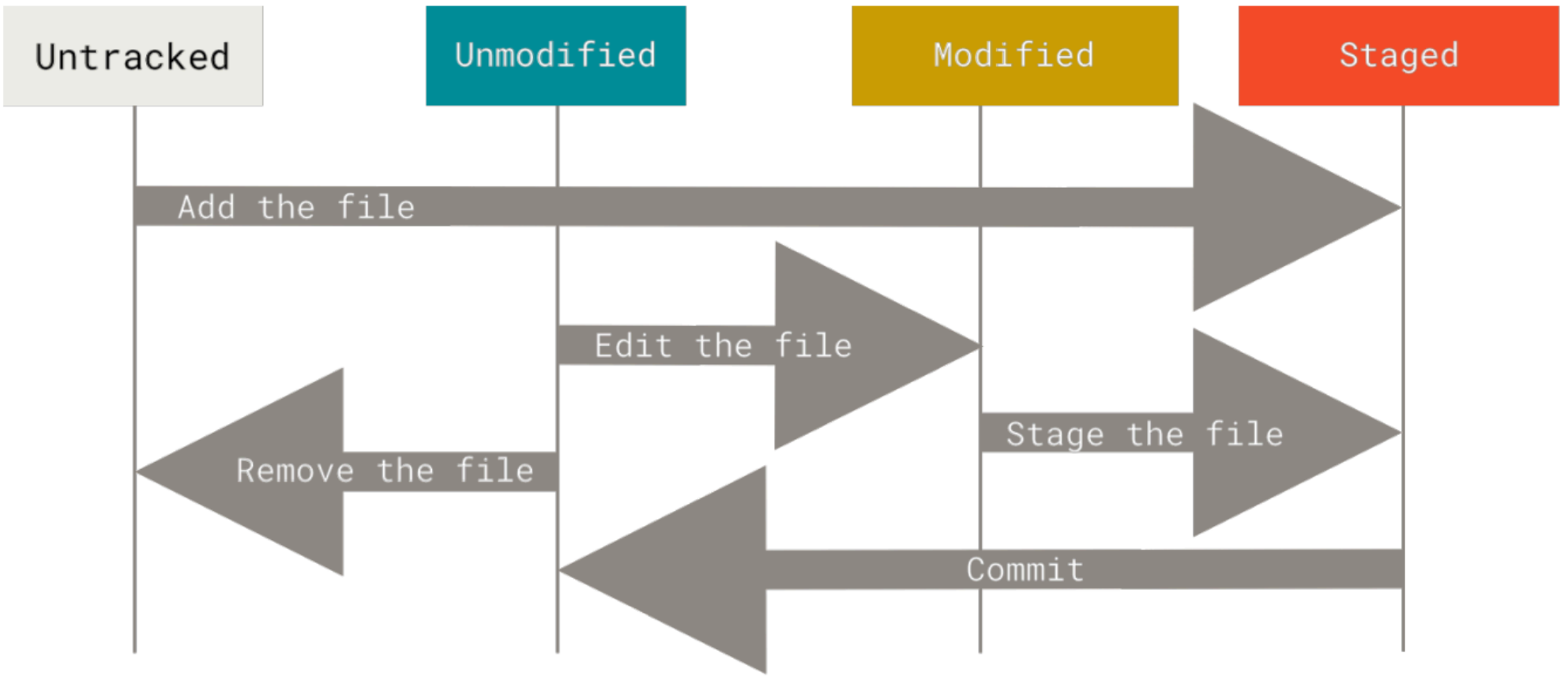


Figure 8. The lifecycle of the status of your files

What did we do? we moved the file from “staged” to “unmodified”.

worked example (4 of 4)

```
% echo "This text will be added to the end." >> foo.txt
% git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   foo.txt

no changes added to commit (use "git add" and/or "git commit -a")
% git add foo.txt
```

Interpretation of step 4

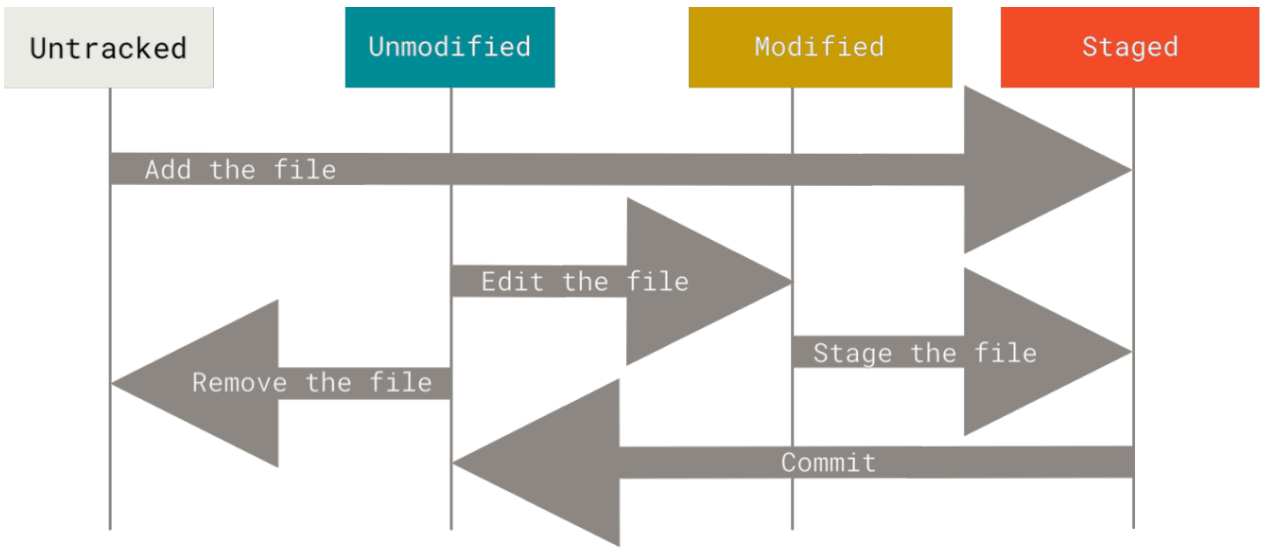


Figure 8. The lifecycle of the status of your files

worked example (4 of 4)

```
% echo "This text will be added to the end." >> foo.txt
% git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   foo.txt

no changes added to commit (use "git add" and/or "git commit -a")
% git add foo.txt
```

Do-Pair-Share

Before starting, run `% git --version` to verify that git is installed. (Installation guide)

Go back through the slides in the worked example and run each of these steps at your command line.

Questions

1. What will we get if we run `git status` at the end of the example?
2. What will we get if we run `git commit -m "my second commit"` and then run `git status`?

Do-Pair-Share Commands Collected Together

```
% git init test
% cd test
% echo "hello world" > foo.txt
% git status
% git add foo.txt
% git status
% git commit -m "create foo.txt"
% git status
% echo "This text will be added to the end." >> foo.txt
% git add foo.txt
```

Helpful git syntax

1. `% git log` prints log of recent commits
2. `% git diff` among files which have changed since last commit, go line-by-line to show changes
3. `% git add .` adds all files (both untracked and modified) to the staging area

Git Ignore I

- You will often have files or filetypes that you want Git to systematically avoid (*ignore*). Examples:
  - Automatically-generated files from compiling Python (`.pyc`)
  - Large data files (this will become very important on future psets)
  - Mac users: `.DS_Store`!
- Create a file called `.gitignore` to tell git which files to ignore
- You must commit your `.gitignore` to the repo

Git Ignore II

Example `.gitignore`:

```
# ignore compiled python
.pyc

# ignore virtual environment packages
.venv/

# ignore pesky auto-created files on macs
.DS_Store
**/.DS_Store
```

See more examples here (link) including for Python

Summary

command	what it does
`git status`	check which files have changed
`git diff`	line-by-line record of changes to tracked files
`git add <filename>`	stage a file
`git commit -m "<message>"`	commit all staged files
`git log`	see recent commits

- Use `.gitignore` to ignore irrelevant files

Do-Pair-Share gitignore (do in lab)

1. In Terminal (Mac)/PowerShell (PC), create a repo named `test`
2. Change directory so you are inside the repo
3. Create three files in a text editor (like VS Code): `small_file.txt`, `large_file.txt` and `.gitignore`
4. Stage `small_file.txt` and then commit it with message "my first commit"
5. Using a text editor, edit `.gitignore` to include the line `large_file.txt`
    - Tip: `.gitignore` is a hidden file. May need to show hidden files in Finder (Mac) or File Explorer (PC)
6. Commit `.gitignore` with a message saying "ignore large file"

Tips:

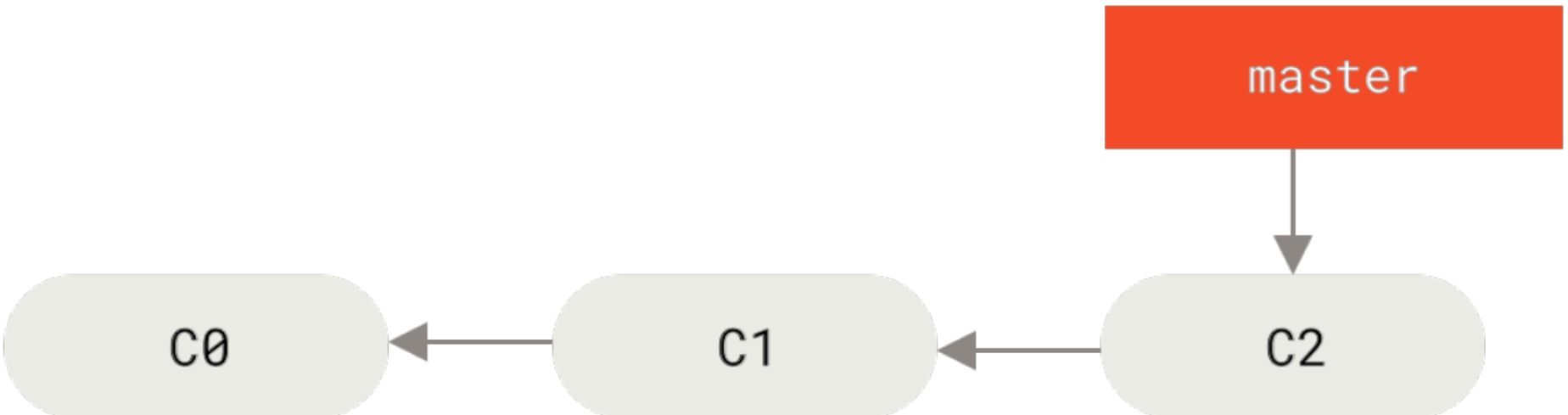
- in between every single step run `git status`, `git diff`, and `git log`
- When you are done, expect to get a message which says "On branch master, nothing to commit, working tree clean"

Track Multiple Versions on Local – Branching (Chapter 3)

Roadmap

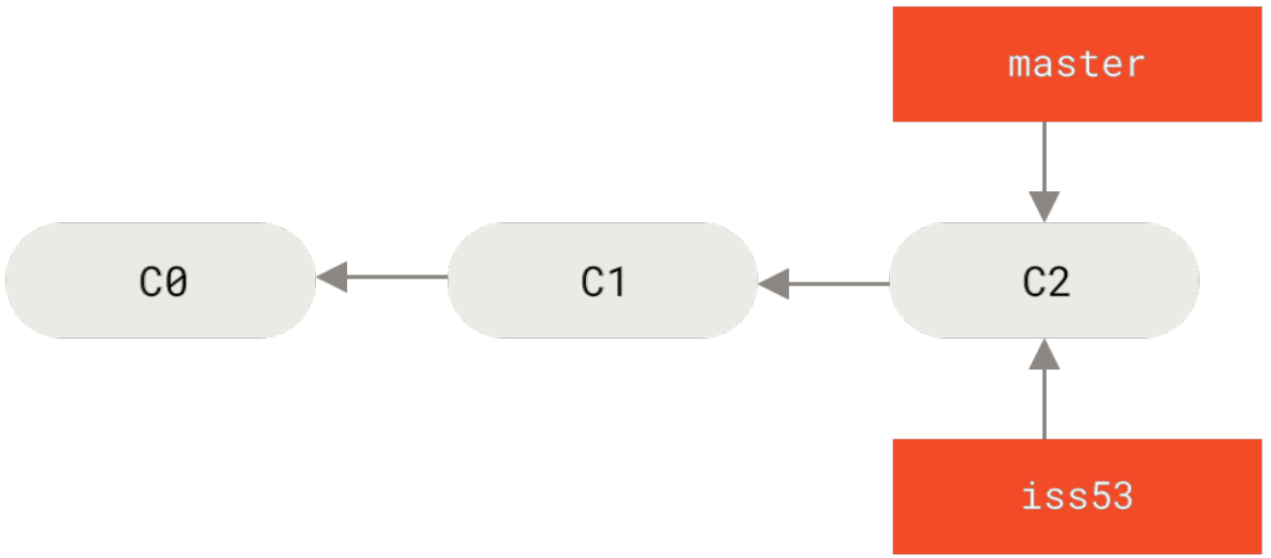
- Branching: overview
- Creating a New Branch
- `HEAD` and Switching Branches
- “Undo”

Branching



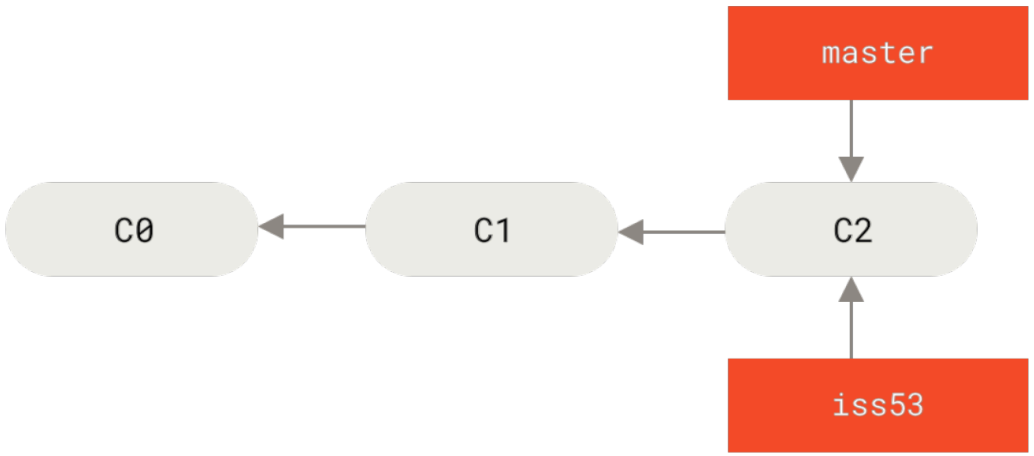
- We will see a lot of images like this, so let's unpack it in some detail
  - Three commits: `C0`, `C1`, `C2`
    - These are horrible names for commits, I hope you never use these!
    - But they are pedagogically quite useful because they convey chronology
  - Every commit (except `C0`) has a parent
  - A "branch" is a chain of commits. Branches: this repo has one branch labeled
    - Here: `master`. FYI: Github.com calls this `main`.
- Build this history using Tinkertoys (same toys used in "git for ages 4 and up" video, invented in Evanston Illinois!)

Creating a New Branch

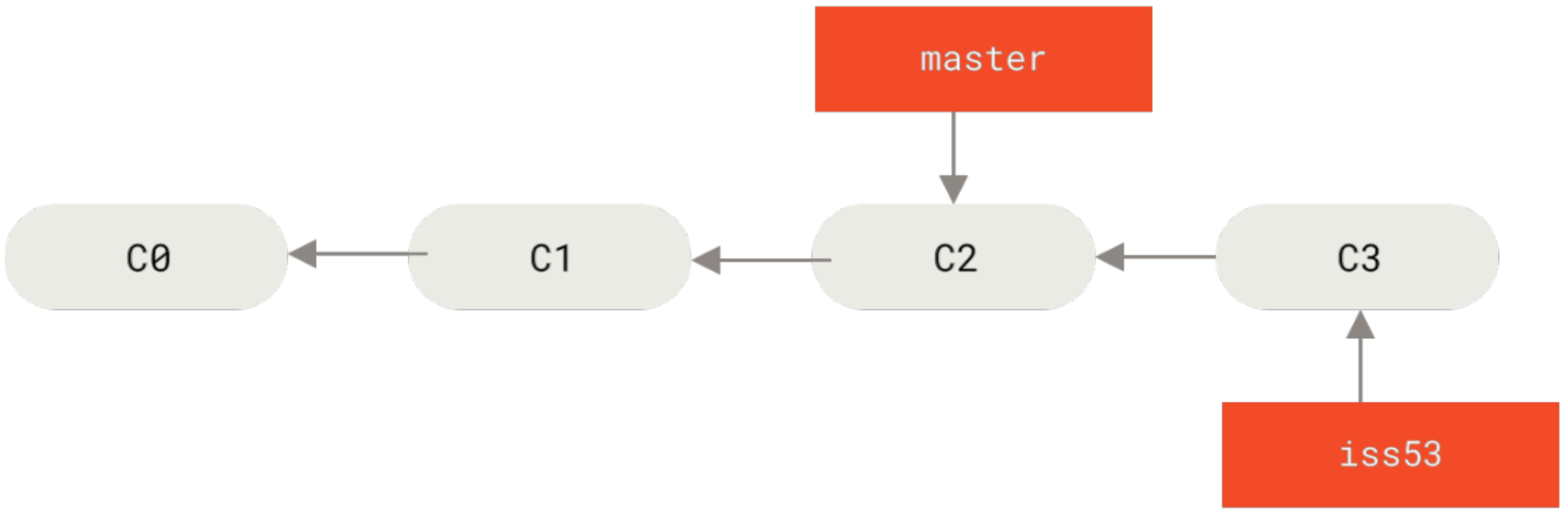


- Say we want to do some exploratory work in a safe environment without affecting the work other team members are doing –> *create a new branch*
- `% git branch iss53` for issue 53. (Update tinkertoy)
- Reminder: “A branch is a chain of commits.” So at the moment both `master` and `iss53` label the chain C0 <- C1 <- C2
- `% git branch` returns a listing of your current branches

Introduce HEAD



Adding to a branch




```
[make some changes to file.py]
% git commit -m "C3: start work on new feature"
```

- Now `iss53` is the chain `C0 <- C1 <- C2 <- C3`.
- `Master` untouched (it is still C0 <- C1 <- C2)

Undo conceptual

From git for ages 4 and up

> Git doesn't throw things away once committed, you can always get back to where you were before

There is **no way** to erase a commit[1]

So… how can I undo my work?

“Undo” options

Scenario	command	gone forever?
haven’t commited yet	`git restore <file_name>`	yes
revise the most recent commit	`git commit --amend`	yes
undo changes from the most recent commit with a new commit	`git revert <commit>`	no
Move `HEAD` back by one commit	`git reset HEAD~1`	no
Move `HEAD` to arbitrary prior commit	`git reset <commit>`	no

Remark: it’s unlikely you will commit (haha) all these scenarios to memory. Instead, use this slide as a helpful reference.

Summary

- Working in branches creates an independent, safe development environment
- Commits only modify current branch, leaving others untouched
- Git doesn’t throw things away, you can always get back to where you were before. How you get there depends on what exactly you want to do.

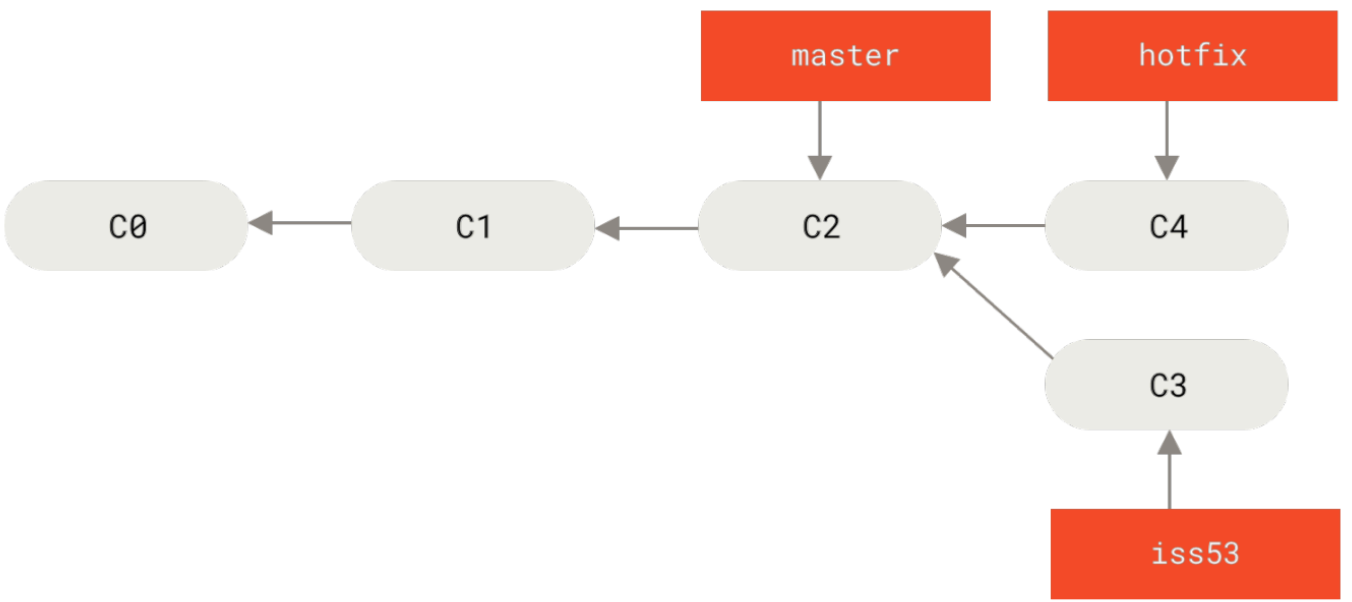
Reconcile Multiple Versions on Local – Merging (Chapter 3)

Roadmap

- Fast forward merge
- Three way merge

Fast forward merge I

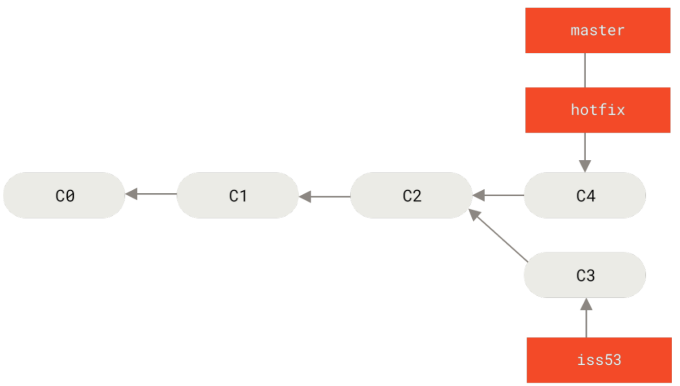
- Your supervisor reaches out with an urgent matter. The dashboard you built is broken.
- You want to pause existing work in `iss53`, switch back to `master`, and make adjustments in a new `hotfix` branch




```
% git switch master
% git branch hotfix
% git switch hotfix
[make some changes to myfile.py]
% git add myfile.py
% git commit -m "commit message for C4"
```

Fast forward merge II

You tested your work and you are ready to put it into production by updating the `master` copy of the code.

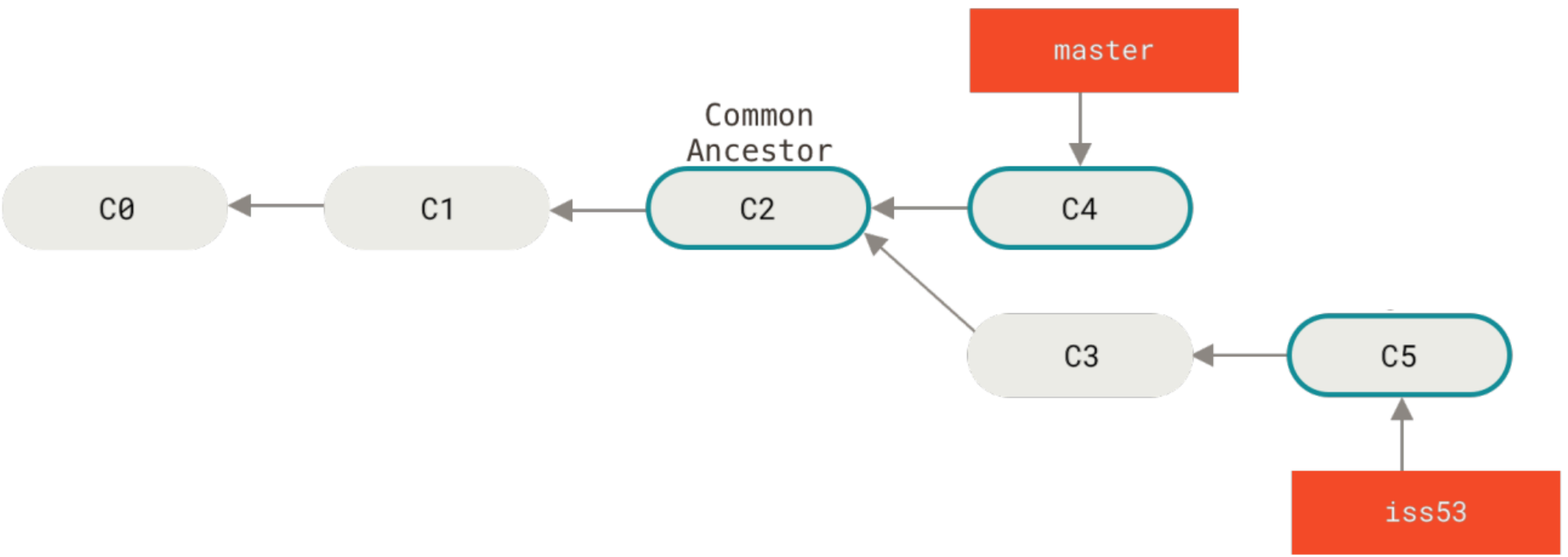



```
% git switch master
% git merge hotfix --ff-only
% git branch -d hotfix
```

1. Navigate to `master`
2. Bring changes from `hotfix` into `master`:
   - `--ff-only` specifies that this is a *"fast forward"* merge
   - update tinkertoy (all we need to do is move the location of the `master` label)
3. You can now safely delete the branch `hotfix` (`git branch -d hotfix`)

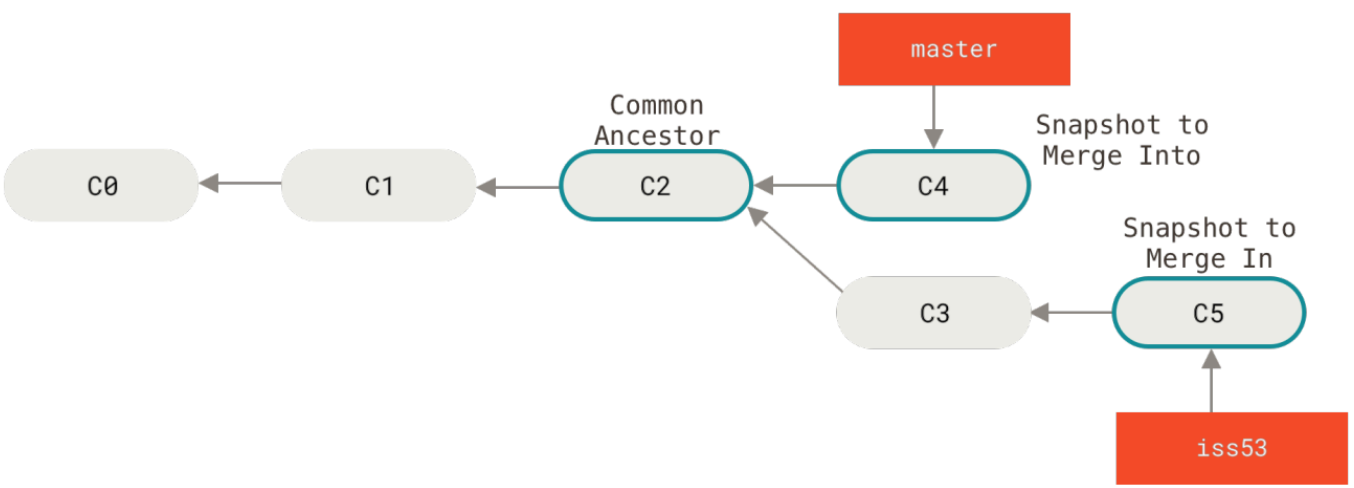
Three way merge I

- After addressing the urgent bug fix, you do some more work on `issue53` and add another commit `C5`.



Three way merge II

- You test the code and it is ready for production so you decide to bring this work into `master` as well (update tinkertoy)



This is called a *"three way merge"* because it relies on three inputs: the *nearest common ancestor* (`C2`), current branch (`C4`),

Three way merge III

You ask git to merge but it refuses!

```
% git switch master
% git merge iss53 --ff-only
hint: Diverging branches can't be fast-forwarded, you need to either:
hint:
hint:   git merge --no-ff
hint:
hint: or:
hint:
hint:   git rebase
hint:
```

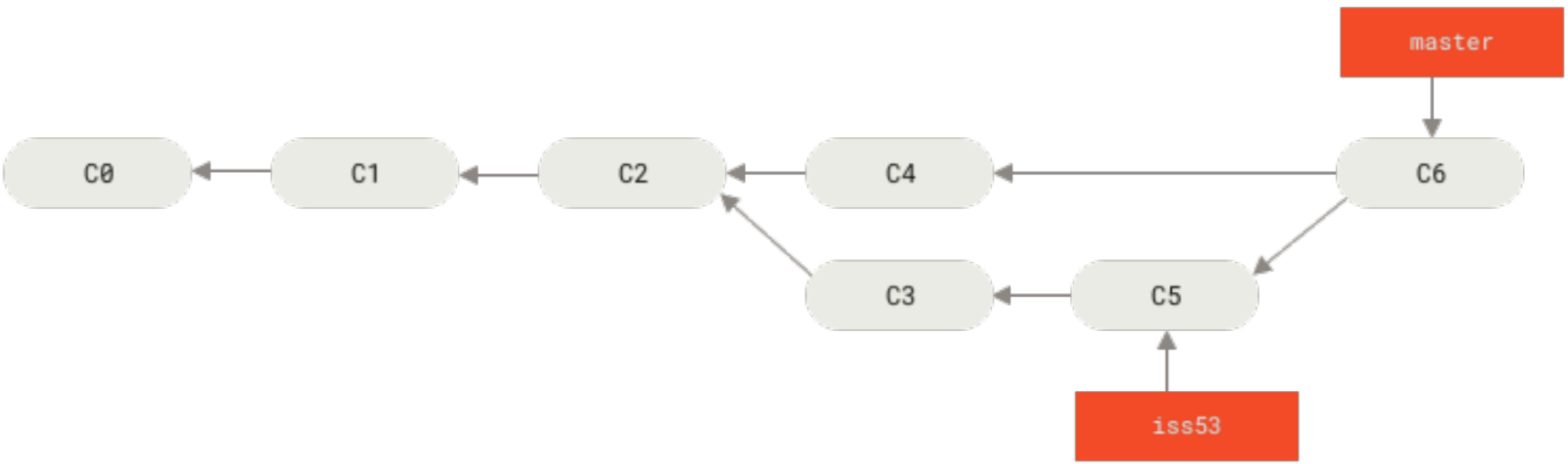
Problem: We need to combine C2, C4, and C5 into a single unified version of the code

Solution `git merge --no-ff`: Git will combine the work for you, creating a new commit `C6`

Remark: `git rebase` not covered in lecture, but you get to try it on the problem set.

Three way merge IV

```
% git merge iss53 --no-ff -m "C6"
Merge made by the 'ort' strategy.
```



Tips on merging

- We have taught `git merge <branch> --ff-only` and `git merge <branch> --no-ff -m "explain what's been merged"`.
- In principle, you can ask git to merge without telling it which type of merge or what is the commit message, but we discourage this. It’s like joining multiple tables without knowing the details of the join (e.g. 1:1 or 1:many)!

Summary

Once work has happened in multiple branches, there will likely be a desire to reconcile down to a single version of the code.

has work happened in multiple branches?	**merge type**	**sample code**
no	fast forward	`% git merge hotfix --ff-only`
yes	three way	`% git merge iss53 --no-ff -m "msg"`

Merge conflicts (Chapter 3)

Roadmap

- Three way merge with a conflict
- Do-pair-share
- Resolving file-level conflicts

Three way merge with a conflict I

- Let’s now switch examples and re-build our tinkertoy
- We will look at a **different** three way merge situation
- We will use the repo merge_example

Three way merge with a conflict II

Original Code

```
1  '''
2  A function that adds x to 3
3  '''
4  def add(x):
5      return (x + 3)
```

Alternate Versions

```
1  '''
2  A function that add
3  '''
4  def add(x):
5      return (x + 4)
```

```
% git switch main
```

```
1  '''
2  A function that add
3  '''
4  def add(x):
5      return sum(x)
```

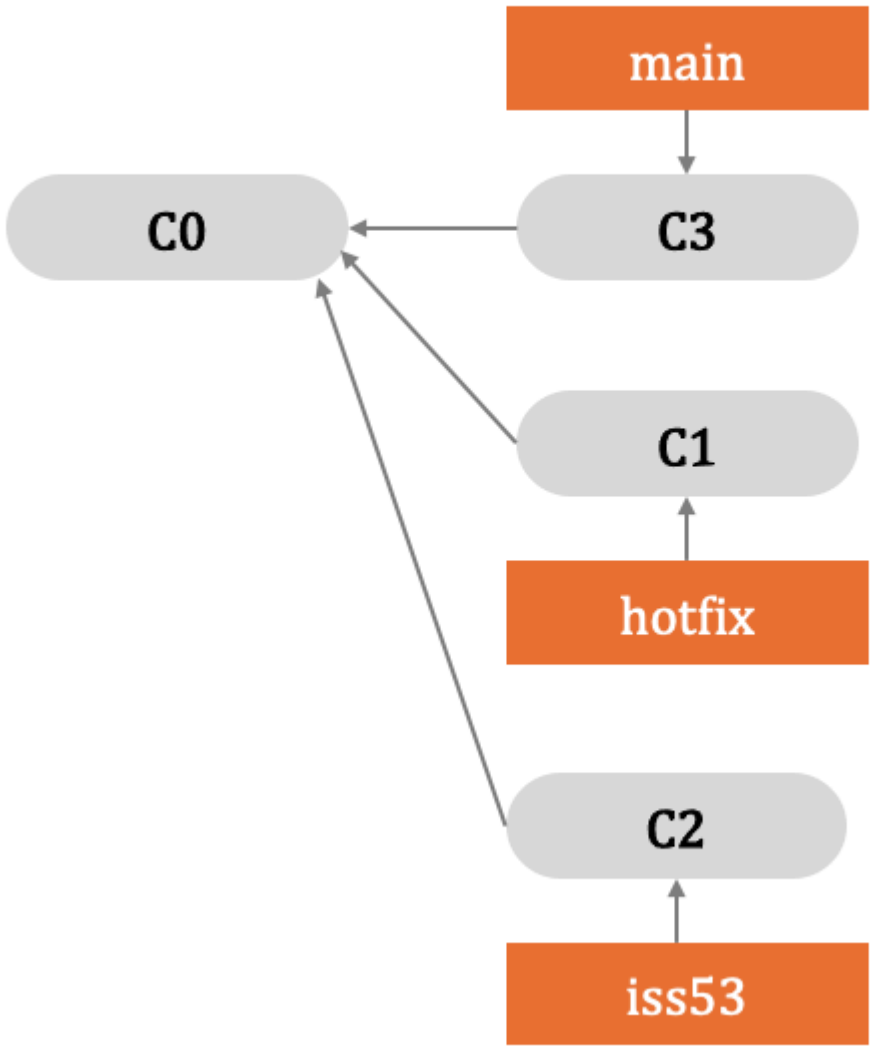
```
% git switch iss53
```

```
1  '''
2  A function that add
3  '''
4  def add(x, y):
5      return (x + y)
```

```
% git switch hotfix
```

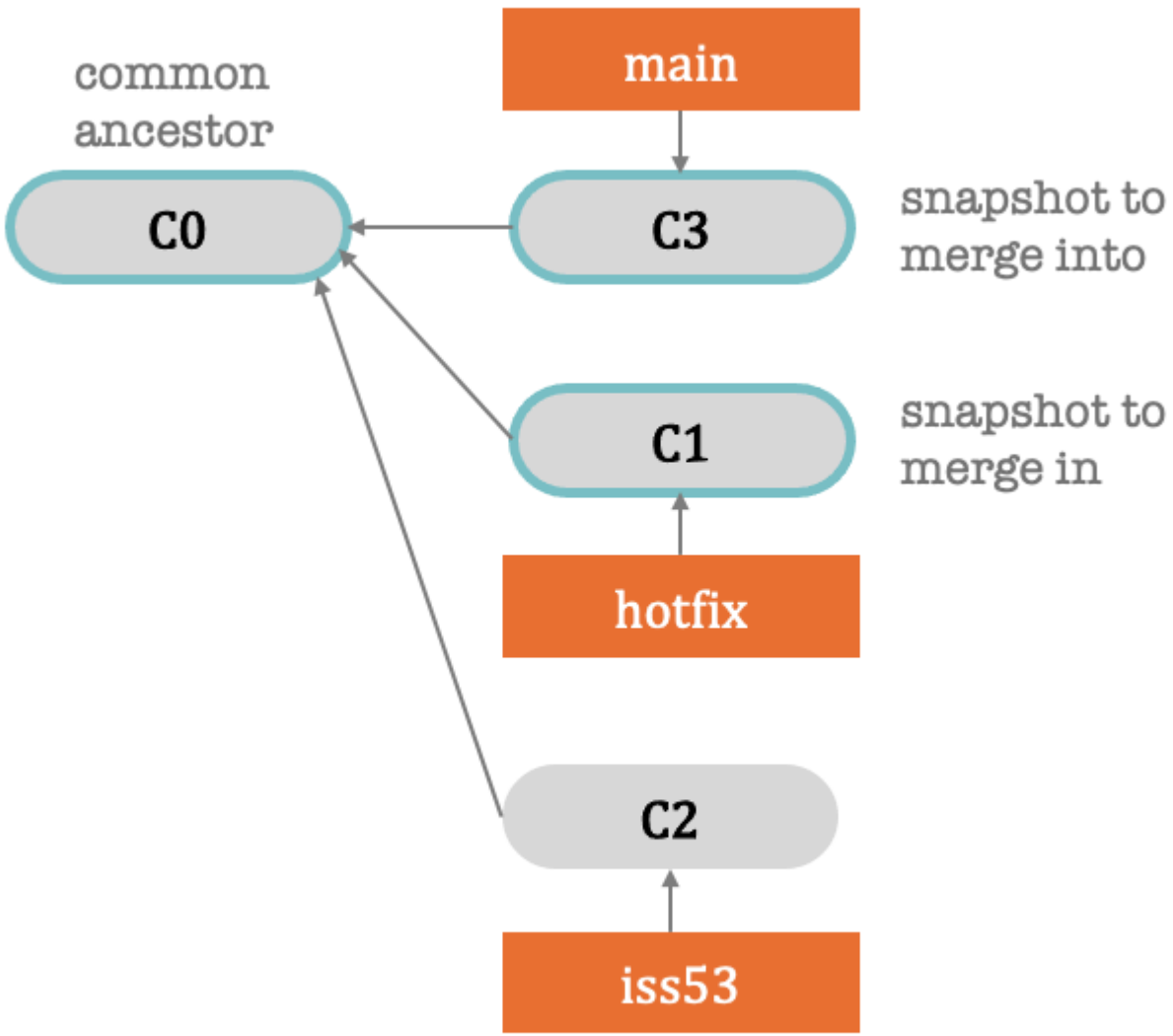
- Running *git switch* command switches to different branches, and each branch contains a different version of *add.py*

Three way merge with a conflict II


main
C0
C3
C1
hotfix
C2
iss53

Three way merge with a conflict III

- We will try to merge hotfix into main



Three way merge with a conflict V

- Now let's try to merge `main` and `hotfix` together
- Doing so will create a new commit, C4

name	branch	content	parent	conflict?
C0	`main`	Create `add.py`, which adds x to 3	none	no
C1	`hotfix`	Change `add.py` to add x and y	C0	no
C2	`iss53`	Change `add.py` to add a list	C0	no
C3	`main`	Change `add.py` to add x to 4	C0	no
C4	`main`	Merge `hotfix` into `main`	C1 & C3	yes!

Three way merge with a conflict VI

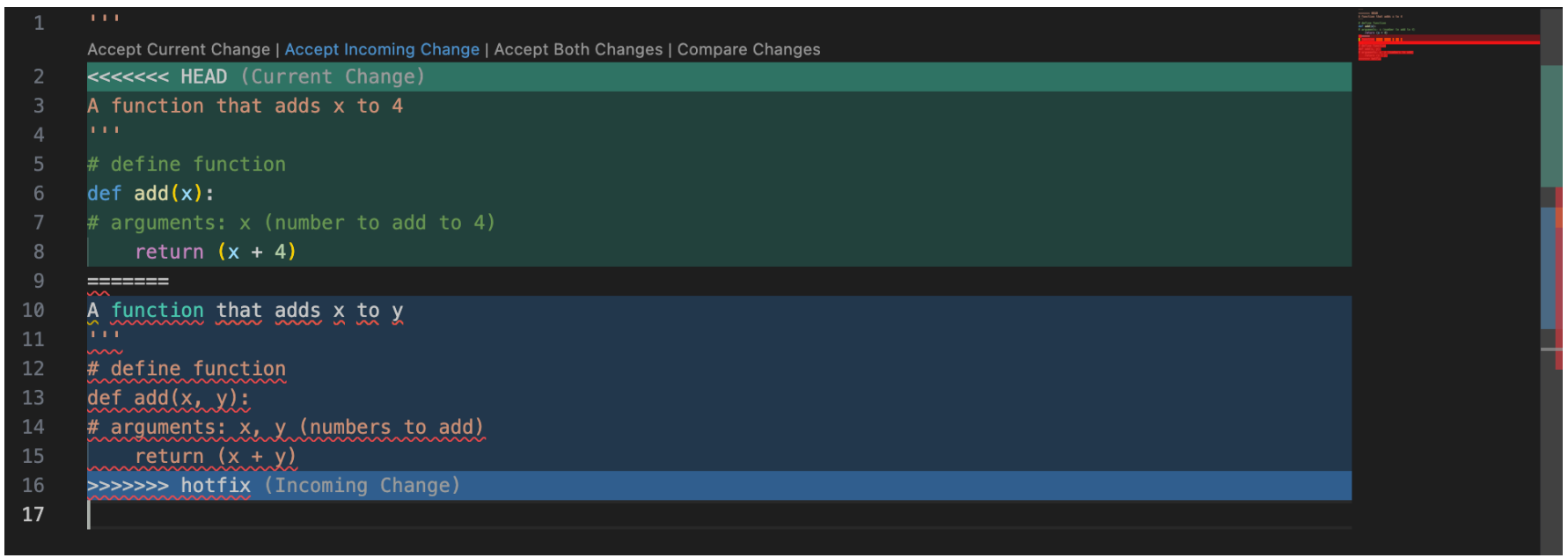
```
% git switch main
% git merge hotfix --no-ff -m "reconcile addition"
```

- Because we have different versions of `add.py` in each branch, merging `hotfix` into `main` causes conflicts!
- Uh oh

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

Three way merge with a conflict VII

- After you type `git merge hotfix`, git will edit your `add.py` file and use the following format wherever it finds a conflict:



- Resolve within Visual Studio Code by clicking “Accept Current Change” or “Accept Incoming change” or “Accept Both changes”.
- Discussion q: which one to use here?

Three way merge with a conflict VI

- In this case, we'll keep what's in `hotfix`
- How `add.py` looks after we've clicked "Accept incoming change": exactly like version in `hotfix`

```
'''
A function that adds x to y
'''
# define function
def add(x, y):
# arguments: x, y (numbers to add)
    return (x + y)
```

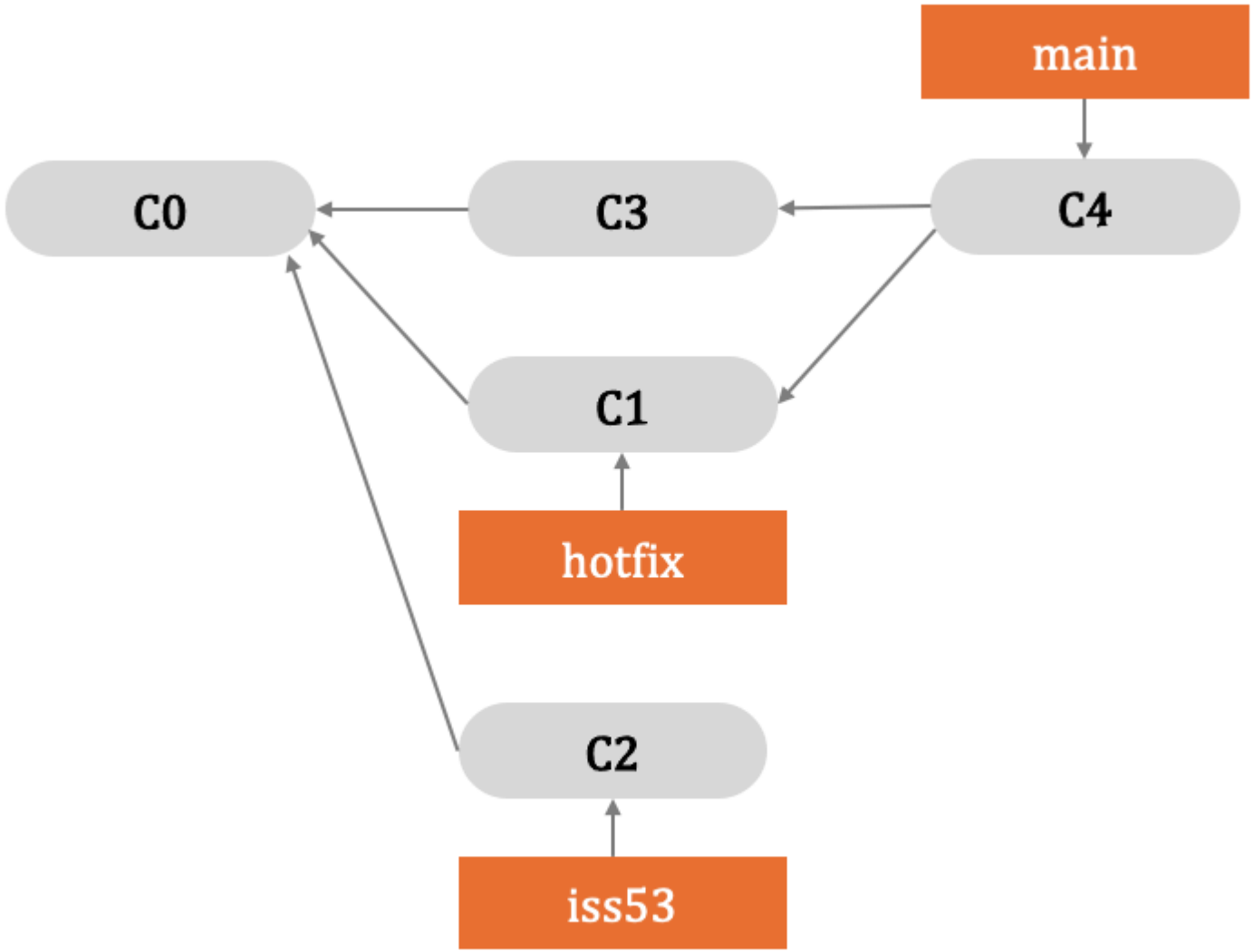
- (Of course, you could have resolved it by "Accepting current change", which would look exactly like `main`!)

Three way merge with a conflict VII

- After resolving, have to add and then commit the merge

```
git add add.py
git commit -m "C4: resolved conflict between main and hotfix and merged"
```

Three way merge with a conflict VIII – image


main
C0
C3
C4
C1
hotfix
C2
iss53

Three way merge with a conflict VIII – ascii

```
C0 <- C3 <- C4
   \  C1 /
   \  C2
```

We are writing the git tree this way since you may need to draw a commit tree on the weekly quiz

do-pair-share: merge conflicts

Resolving file-level conflicts I

- For some file types, line-by-line adjustments will not be possible
  - E.g. PDFs, images

Resolving file-level conflicts II

- Consider a repo with two versions of `qmd` here
  - Branch *iss1* has a version of the file *Example.pdf*
  - Branch *main* has another version of *Example.pdf*
- How to resolve the conflict when merging *main* into *iss1*?
  - Because these are PDFs, we can't open them in a text editor to resolve the merge
  - Instead, we have to declare which of the two files we will keep

Resolving file-level conflicts III

- Solution: `git checkout` command with
  - `--theirs`: keep file from incoming branch (`iss1`)
  - `--ours:` keep file from current branch (`main`)

```
% git merge iss1
% git checkout --ours Example.pdf
% git add Example.pdf
% git commit -m "C5: Merging main into iss1"
```

Summary

- When branches have divergent commit histories, you may have to manually resolve conflicts
- Some conflicts must be resolved by keeping files from one branch or another

Reconciling Your Version with a Remote (Chapter 2.5 and 3.5)

Motivation and roadmap

- Up to this point, everything we've been doing is local
- But for collaboration, you will be working off a shared directory
- And that shared directory is usually hosted on Github

Roadmap

- Create an online repo
- Upload commits from local repo to online repo (`push`)
- Download commits from remote repo to local repo (`fetch`)
- Reconcile commits between downloaded and local (`pull`)
- list all branches
- pull requests
- fork

Create a new repo online

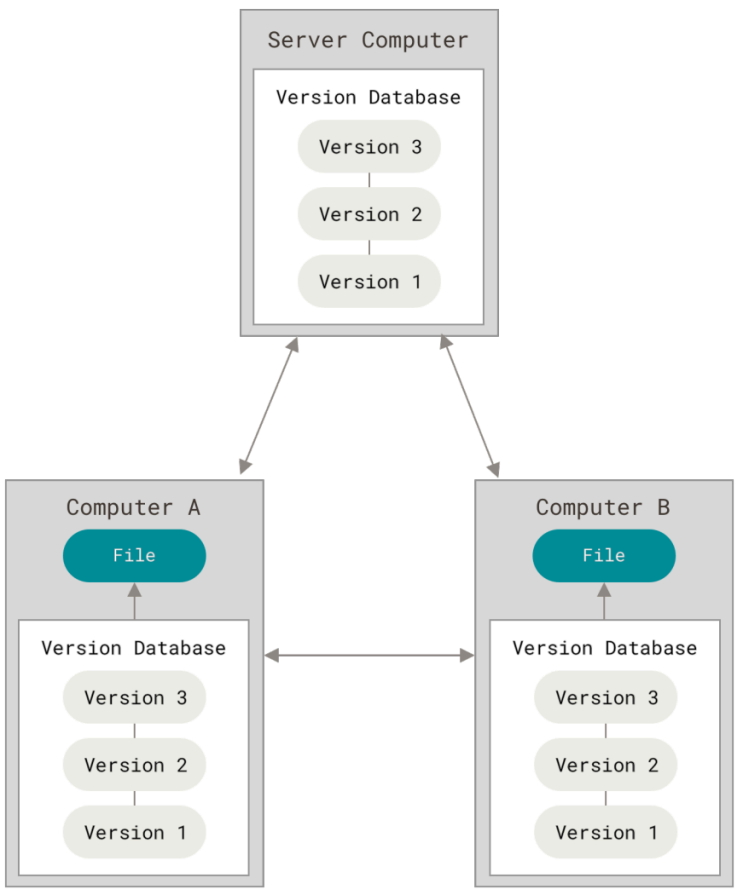
https://github.com/new and then clone to your computer

Choices

1. Public or private?
    - Public: your final project, anything else that demonstrates your skills
    - Private: sensitive data, pset solutions
2. License – document that tells others what they can/cannot do with your code.
    - Ganong's lab uses the MIT license – allows others to use, modify, and disribution code with attribution
    - Key thing is to specify **a** license, otherwise people won't be able to use your code!

Recall: “distributed” version control

- Each computer fully mirrors the remote repository, including its full history



push

```
% git push  <branch_name>
```

- Upload (committed) changes from local to remote repository
- Until you do this, no other users will be able to see/access your commits

fetch

```
% git fetch
```

- Download other users' pushed commits from remote repository to your computer – but doesn't merge them in yet
- This command does not modify any of your existing work.

pull

- *git pull* equivalent to *git fetch* followed by *git merge*

```
% git pull  <branch_name>
```

- Automatically *fetches* branch from the remote repository, then *merges* that branch into your current branch
- If there are reconcilable changes in both places, git will create a reconciling merge commit and ask you to confirm the content of the git message.
- If there are irreconcilable changes, then you are in the world of the last section of lecture where you need to do some reconciliation.

see *all* branches

- list all branches, including branches in the remote repository which are not on your computer

```
% git branch -a
```

Note: we already had this as a “tip” in the do-pair-share

what happens if you push and there are also changes in the remote?

You will see in Terminal/Powershell:

```
Updates were rejected because the remote contains
work that you do not have locally. This is
usually caused by another repository pushing to
the same ref. You may want to first merge the
remote changes (e.g., hint: 'git pull') before
pushing again.
```

Pull Requests

- A pull request is a way to *propose* changes from your branch to be merged into the main project/branch
- Doesn't `git merge` already do this? Yes, but pull requests allow teammates to review your code before it gets merged.
- Example: an open pull request for `merge_example` repo (here)

Compare main and iss53 via Pull Request #1

Edit | <> Code

Open **whpennington** wants to merge 1 commit into `main` from `iss53`

Conversation 0 | Commits 1 | Checks 0 | Files changed 1 | +3 −3

whpennington commented now

No description provided.

C2: Change add.py to add list on iss53 7db9958

This branch has conflicts that must be resolved

Use the web editor or the command line to resolve conflicts.

Resolve conflicts

Conflicting files

`add.py`

Merge pull request | You can also open this in GitHub Desktop or view command line instructions.

Add a comment

Write | Preview

Add your comment here...

Reviewers

No reviews

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

None yet

Notifications Customize

Unsubscribe

You're receiving notifications because you're

Pull Requests

- Pull requests allow for line-by-line comparison of changed files on Github.com
  - Click “+” to add a comment (show in browser)
  - They also identify merge conflicts

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks or learn more about diff comparisons.

base: main ← compare: iss53 ✕ **Can't automatically merge.** Don't worry, you can still create the pull request.

Discuss and review the changes in this comparison with others. Learn about pull requests

Create pull request

1 commit | **1** file changed | **1** contributor

Commits on Sep 26, 2024

C2: Change add.py to add list on iss53

whpennington committed 3 weeks ago

7db9958

Showing **1 changed file** with **3 additions** and **3 deletions**.

Split | **Unified**

6 add.py

```
...  ...   @@ -1,7 +1,7 @@
 1    1    '''
 2         - A function that adds x to 3
      2    + A function that adds a list of numbers together
 3    3    '''
 4    4    # define function
 5    5    def add(x):
 6         - # arguments: x (number to add to 3)
 7         -     return (x + 3)
      6    + # arguments: x (list of numbers to be added)
      7    +     return sum(x)
```

`git merge` vs pull requests

When should I use each?

- Use pull requests when you want to preview a change
- Use pull requests when you want someone else to review the change
- Otherwise, just use `git merge`

`fork`

We have asked you a few times now to fork, without explaining what it actually is.

A fork is a remote copy of an entire remote repo.

Why fork? Experiment with and modify someone else's public code.

This is useful in this class because you have a fork of the student repo. We will keep pushing commits to the public student repo and then you can pull those commits into your individual fork. You can also add your own code to the individual fork without messing up the class-wide repo.

This workflow is the heart of open source development.

Summary

- *push* sends changes from local to remote
- *fetch* brings changes from remote to local
- *pull* brings changes from remote to local and tries to reconcile via merge commit
- *pull requests* provide a “trial run” of merges that is visible to collaborators
- fork creates a remote copy of a remote repo

Final Remarks

- Git is admittedly confusing!
- Now that you know how it works, you have to start using it to get comfortable using the commands
- Professor Shi has a printed version of a Git cheatsheet (`atlassian_cheatsheet.pdf`) taped on the wall in my office. See also this cheatsheet