UCLA CS35L

Week 8

Monday

Reminders

- Assignment 7 due this Friday (5/22)
- Assignment 8 due next Friday (5/29)

- Week 10 Assignment, first presenters are this Wednesday
  - Ty Koslowski + William Randall
- Reach out to me if:
  - You need to send in a recording due to timezone issues making it hard to present live
  - Your partner has not responded to you about preparing for the presentation/report

- Anonymous feedback for Daniel
  - https://forms.gle/tZwuMbALe825DBVn8

Version Control

What is Version Control

- A type of software tool that helps a team manage changes to their source code over time.
- Usually has features like:
  - All changes are stored, and can be rolled back or diff'ed if there is a mistake
  - Prevent concurrent work on the same code from causing problems

History of other VC tools

- Started off with tools that locked a file while a dev was working on it
    - Protected conflicts, but stopped devs from working in parallel
- Later Centralized VC became popular – SVN and CVS
    - SVN uses one central repository to manage all security and controls
    - SVN relies on user’s being always connected to the server, so that way all files are always centrally accessible and up to date
    - All changes done are merged into the “production” line.

Git

- A distributed version control system – also Free and Open Source
- Created in 2005 by Linus Torvalds
  - Was originally designed to work his projects which had a distributed development structure
- Became popular not long after mainly because of:
  - Distributed offline development was possible
  - Branching and Merging was more flexible
  - The “staging area” architecture
  - Github

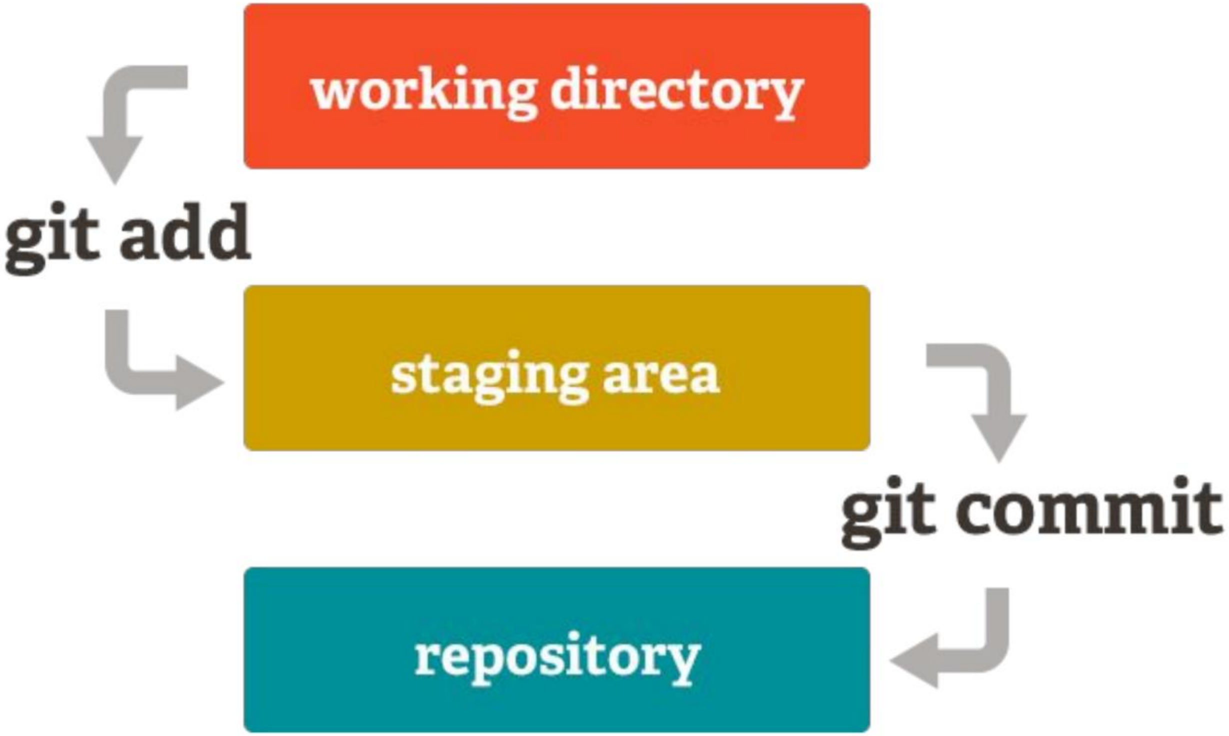
Git

Git vs Github

- Git is the software version control tool.
  - We will focus on this and its architecture
- Github is a hosting service for the server-side portion of Git. Also added features like:
  - Public vs Private Repositories
  - Pull Requests
- There are other Git hosting services as well – BitBucket, GitLab, etc

Git Architecture – Dev Environment

- Made up of 2 main parts
  - Remote Repository
  - Development Environment (Local to your machine)
    - Working Directory
    - Staging Area
    - Local Repository



Local vs Remote Repositories

- Remote Repo
  - Acts as a centralized serve
  - You and team members will **push** changes to server and **pull** updates
  - Technically optional
  - You typically interact with this only an as-needed basis
- Local Repo
  - Is on your local computer
  - Contains all of the files and their commit histories (changes)
  - Most typically will work directly on this

git init – Creating a Repository

```
git init
```

- Indicates that the current folder is the top level of the git repository
- Everything within it can now be tracked
- Note – a new folder “.git” will be created to contain version control information

git clone – Copy an existing repository

```
git clone <someUrl>
```

- Used to copy down an existing repository to a new directory.
  - Typically used to copy a remote repo to your local machine
- Example workflow
  - Create remote repo on GitHub
  - Run `git clone <someURL>` to copy to your selected directory

git status – See current status

```
git status
```

- Shows the current tracking status of all files
    - Modified files – Files that are being tracked, and have been modified since the last commit
    - Untracked files – Files are that are not being tracked in git
    - Red vs Green – Files that are green are in the Staging Area, ready to be committed

git add – move file to staging area

```
git add <files>
```

- You move the selected files and all of its changes to the staging area
- Files can either be
  - Untracked files
  - Tracked files that have been modified
- Note at this point, the change is staged but not yet committed

git commit – create a commit (snapshot)

```
git commit –m “some message”
```

- A commit is a snapshot of all the files at the current project stage.
  - This is what makes a version control tool.
  - Everyone maintains a full record of all the commits to their local repo
  - You can rollback to any previous commit if desired
- All commits are stored with a Hash value which makes them uniquely identifiable

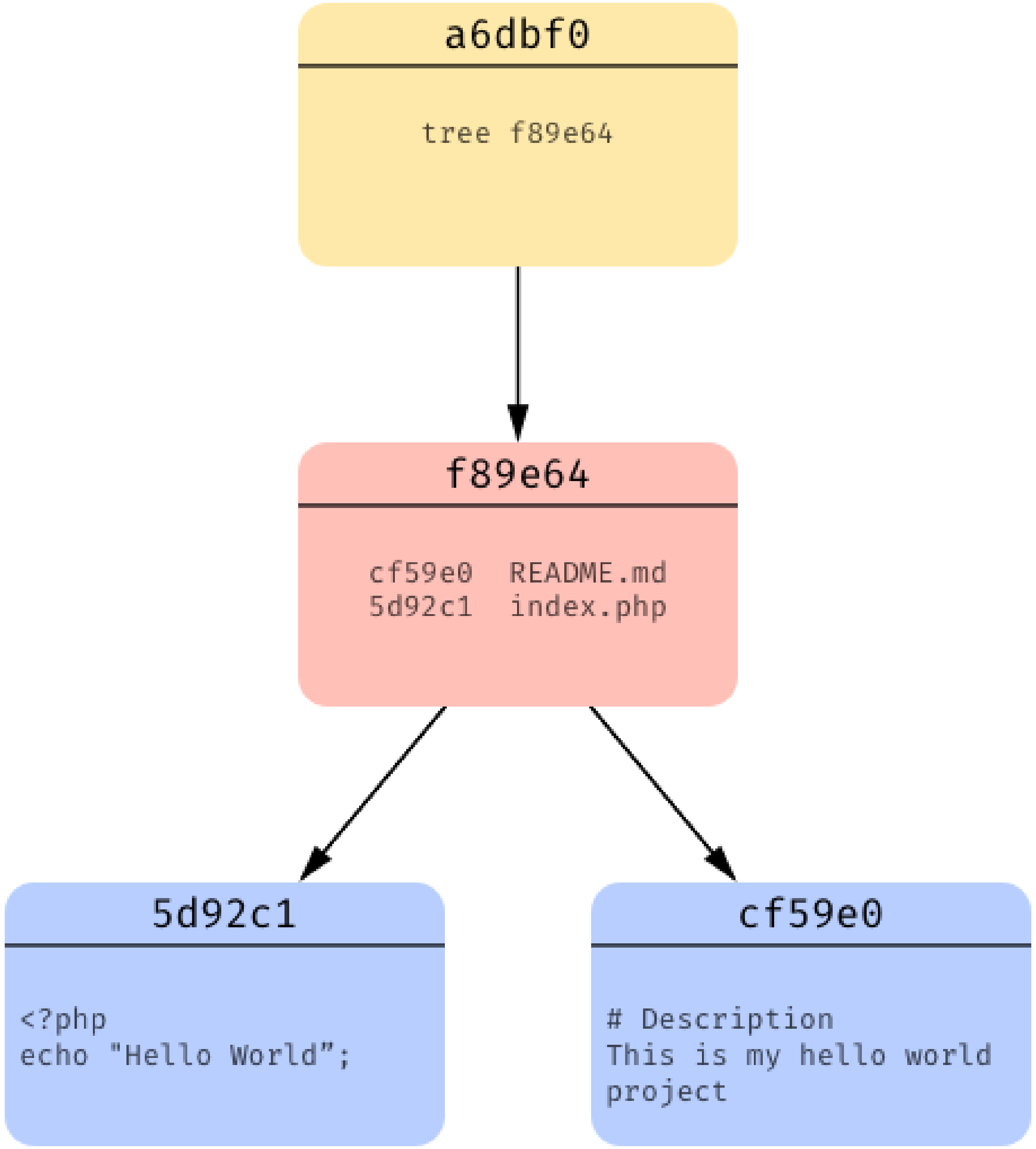
git diff

```
git diff [--staged]
```

- Show in diff file format, all current changes
- By default, only shows changes in the working directory. To show changes in the staged area, use the optional **--staged** option

Commit Structure

- Commits are Snapshots, not just the diffs in between files
  - Makes it easier to move in between commits
  - Still tries to be efficient – doesn’t snapshot files that have not changes and will pack loose objects together
- Commits are identified by a Hash Value
- Commit points to a tree object
- Tree object points to Blob objects (actual files)



a6dbf0

tree f89e64

f89e64

```
cf59e0  README.md
5d92c1  index.php
```

5d92c1

```
<?php
echo "Hello World";
```

cf59e0

```
# Description
This is my hello world
project
```

git log

To see details of the entire commit history

- `git log`
  - Show general details
- `git log --stat`
  - Show stats on file changes
- `git log -p`
  - Show full details on each commit change with diff format
- `git log --pretty=oneline`
  - condense git log output to one line with just hash #

git tag

```
git tag
```

- Lists all the current tags in the directory

```
git tag –a <tagName> -m “tagMessage”
```

- Creates an annotated tag for future safekeeping

- Sometimes it is helpful to mark certain points in the commit history as important and not to be changed.
  - Usually done for version releases

git push

```
git push
```

- Push all of your currently committed changes to a remote repo
- Note if you don’t have a remote repo, can add with
  - `git remote add <name> <someURL>`
  - `git remote add origin https://github.com/myUsername/myRepo`

git fetch and pull

```
git fetch
git pull
```

- Both commands used to pull information from the remote repo
    - **fetch** retrieves metadata information only (what files have changed, etc)
    - **pull** retrieves metadata information and copies any files changes as well
        - Actually runs two commands, git fetch and git merge.

Ex. Very simplified single user git workflow

- Create a git repository on a hosting service like Github
- Clone that git repository to SEAS server and local computer
  - git clone
- Work on device and record changes bit-by-bit
  - git add
  - git commit
- When done on one device, synchronize to the remote repo
  - git push
- When changing to the other device, synchronize to latest copy
  - git pull

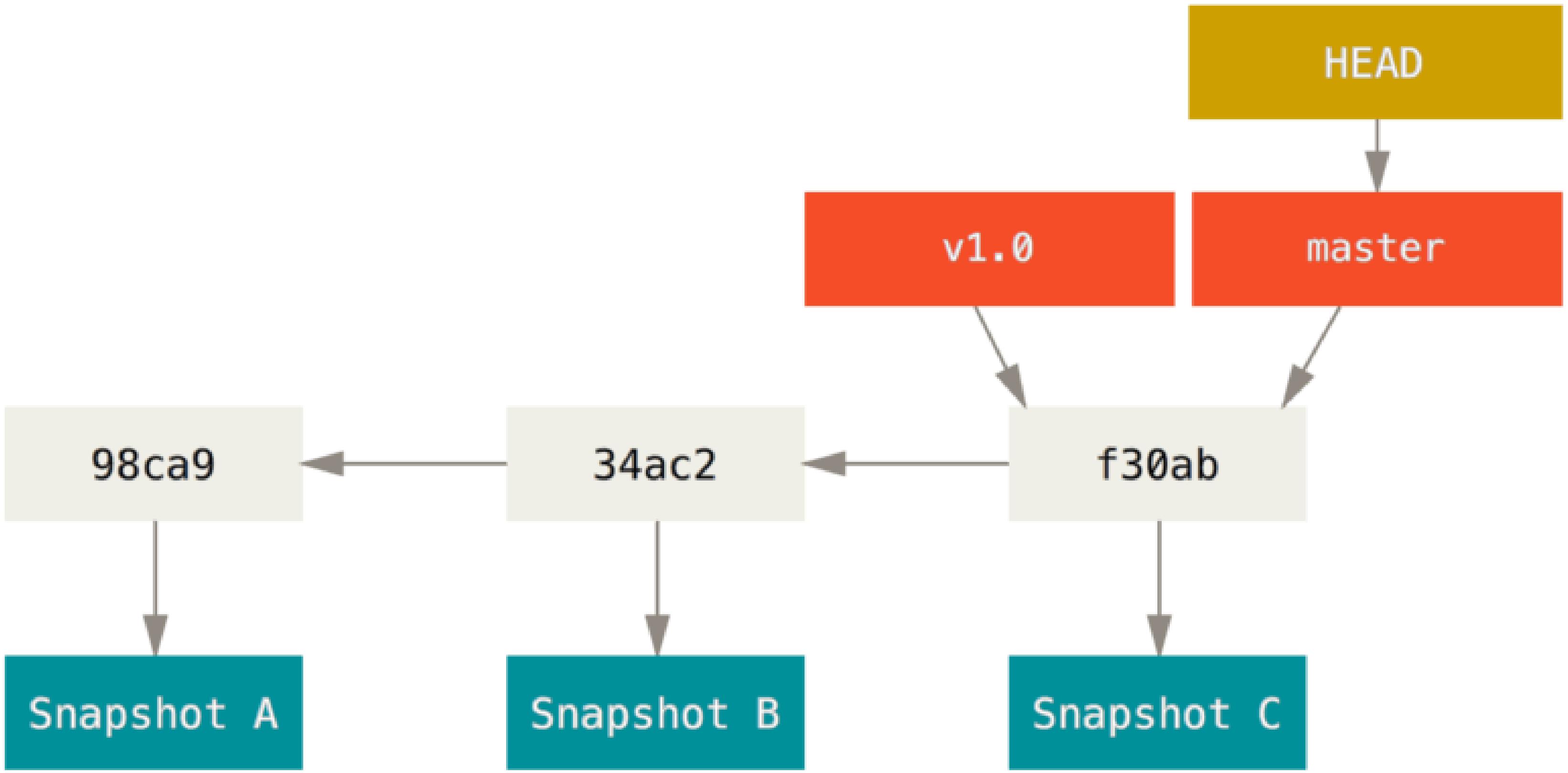
Branches and Merging

Branches

- So far we have only been working on one main trunk
- But the big advantage of version control is to work on separate branches (that don’t affect the original) and are merged in when ready.
- Examples
  - There is a “master” main branch of all working code
  - I make a branch to write the sorting feature and add it in
  - Someone else makes a branch to fix a bug with how master reads input

HEAD Pointer and Branches

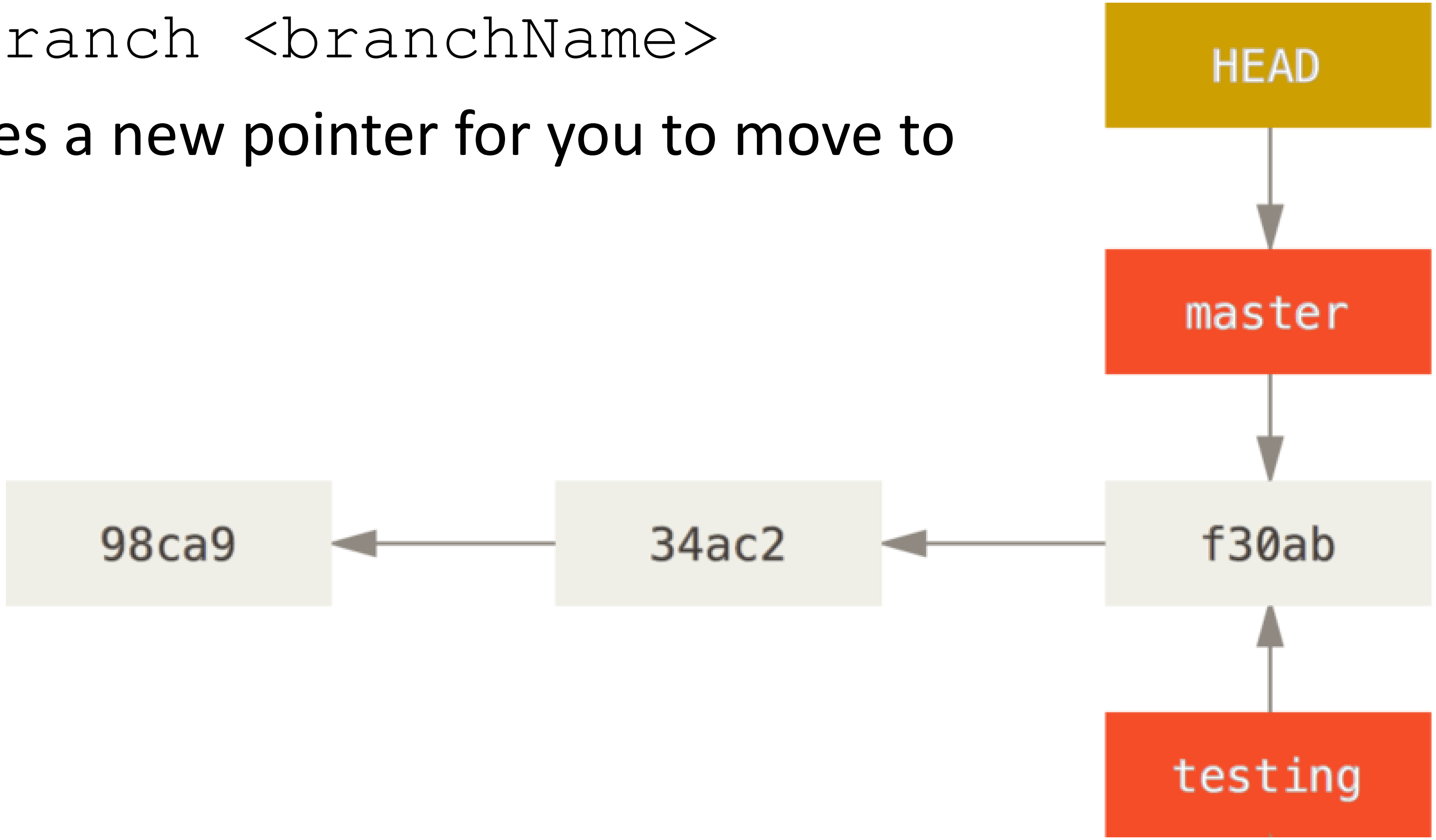
- The HEAD Pointer points to the current commit you are on.
- Each branch contains a pointer to the current commit that it is on.
- master is the first branch usually created, so HEAD starts by pointing at the latest commit in master


HEAD
v1.0
master
98ca9
34ac2
f30ab
Snapshot A
Snapshot B
Snapshot C

Create a branch

```
git branch <branchName>
```

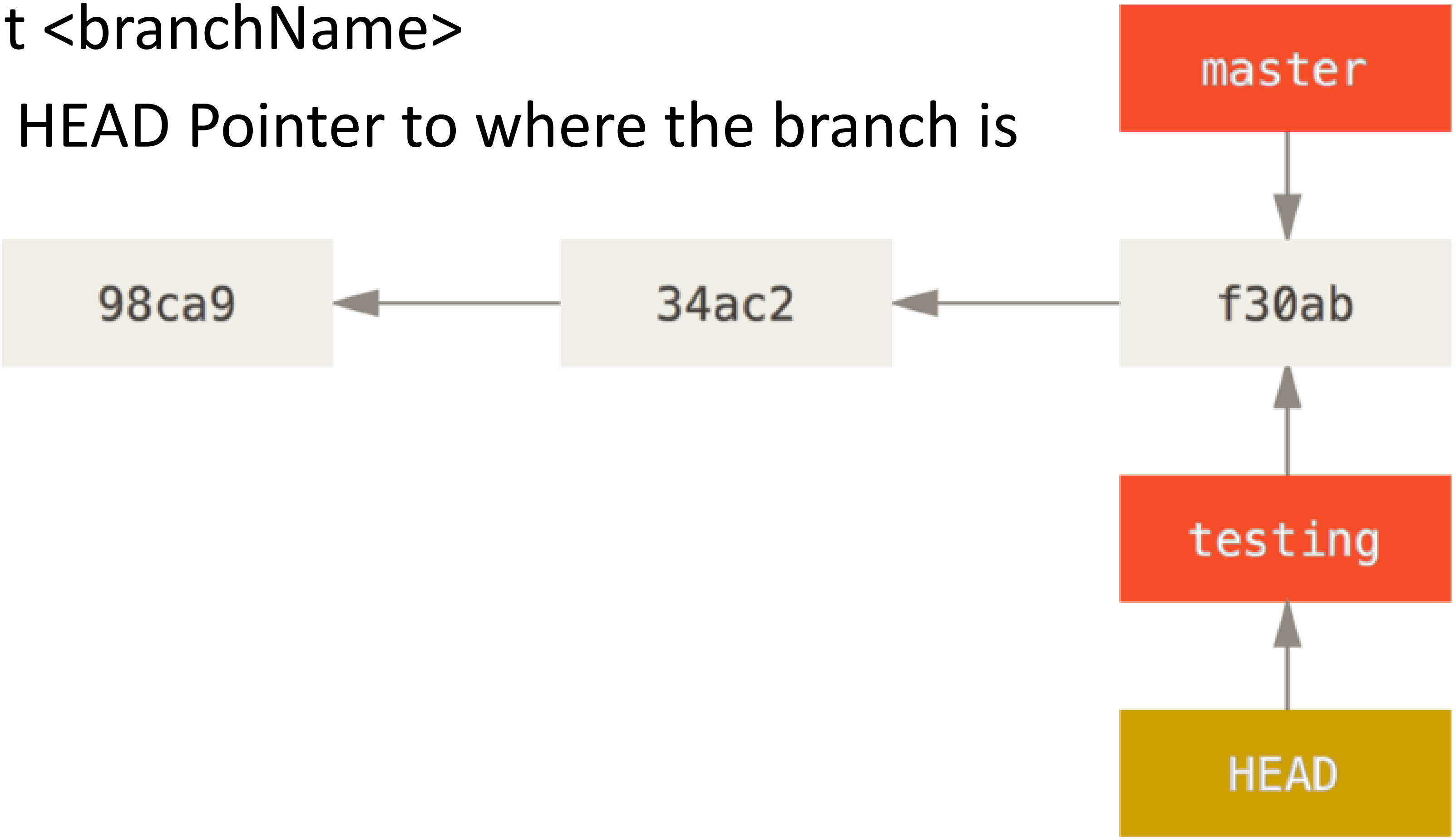
- Creates a new pointer for you to move to



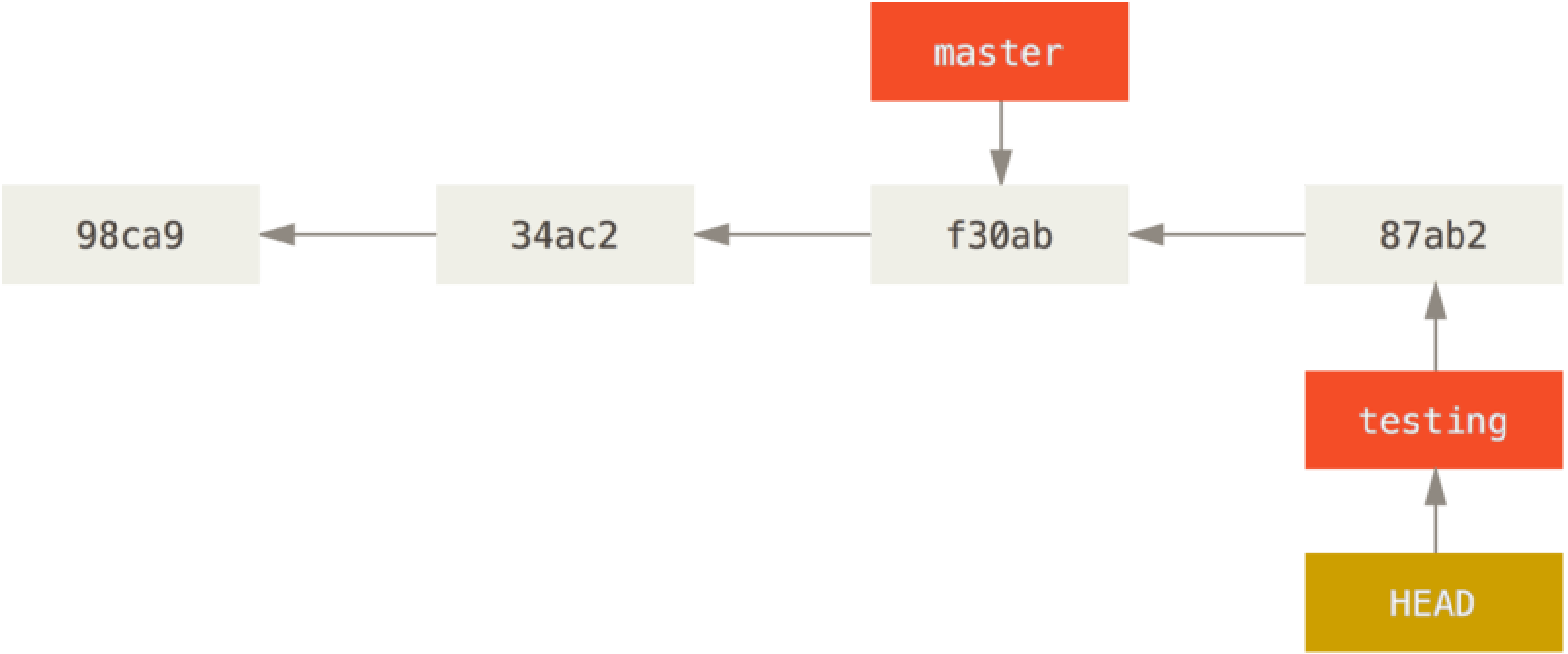
git checkout – switching branches

git checkout <branchName>

- Move the HEAD Pointer to where the branch is



Create commits on Branch



git checkout - Variations

- `git checkout –b <branchName>`
    - Will create the branch if it does not exist
- `git checkout –b <branchName> <remote>/<branch>`
    - Ex. git checkout -b testing origin/testing
    - Will create a local branch, copied from and tracking the remote branch specified

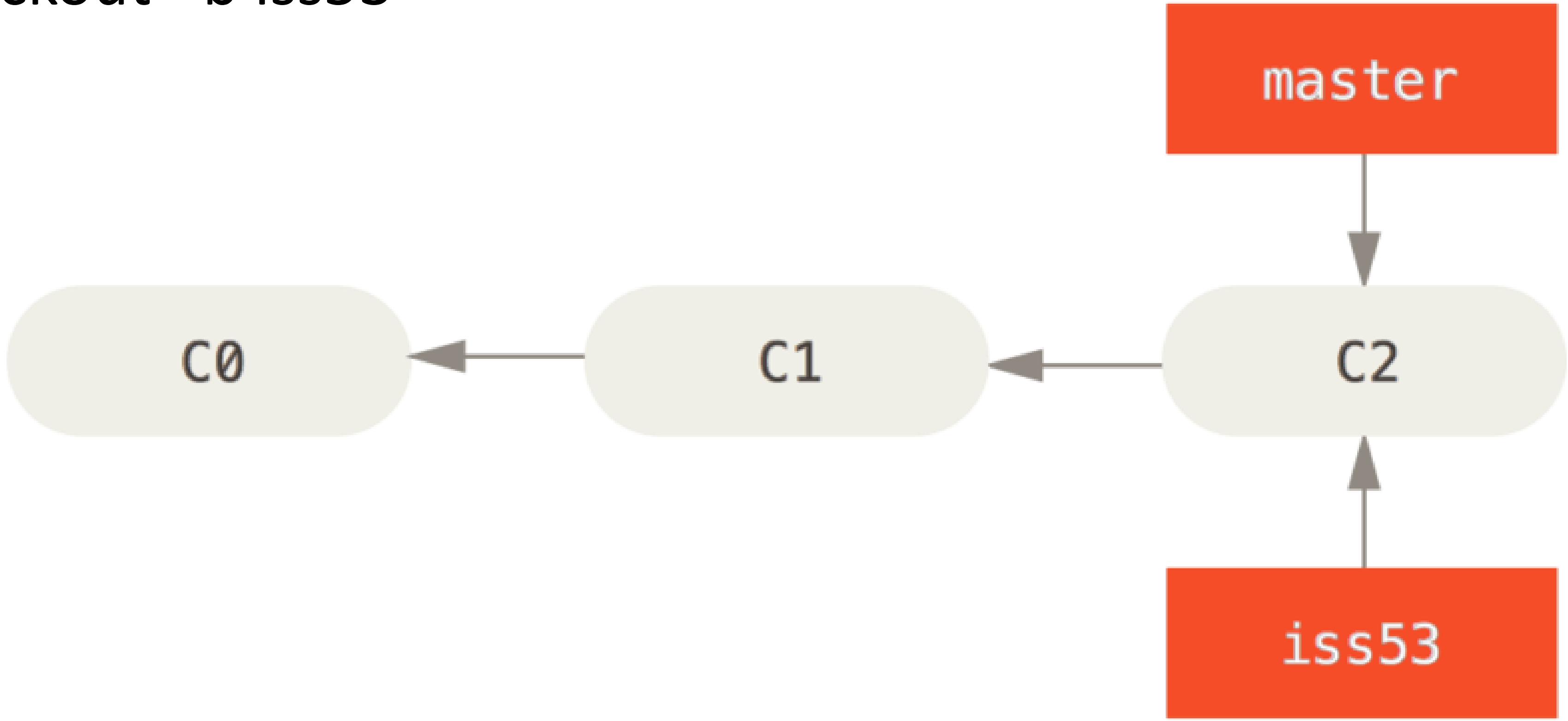
Merging

```
git merge <branchName>
```

- At some point, you are done with your branch and want to merge it into the main line of code.
- Command above will merge the named branch into the current branch you are on.
- Two main types of merges
  - Fast-Forward Merge
  - Three-way Merge
- Will look at examples in next slide

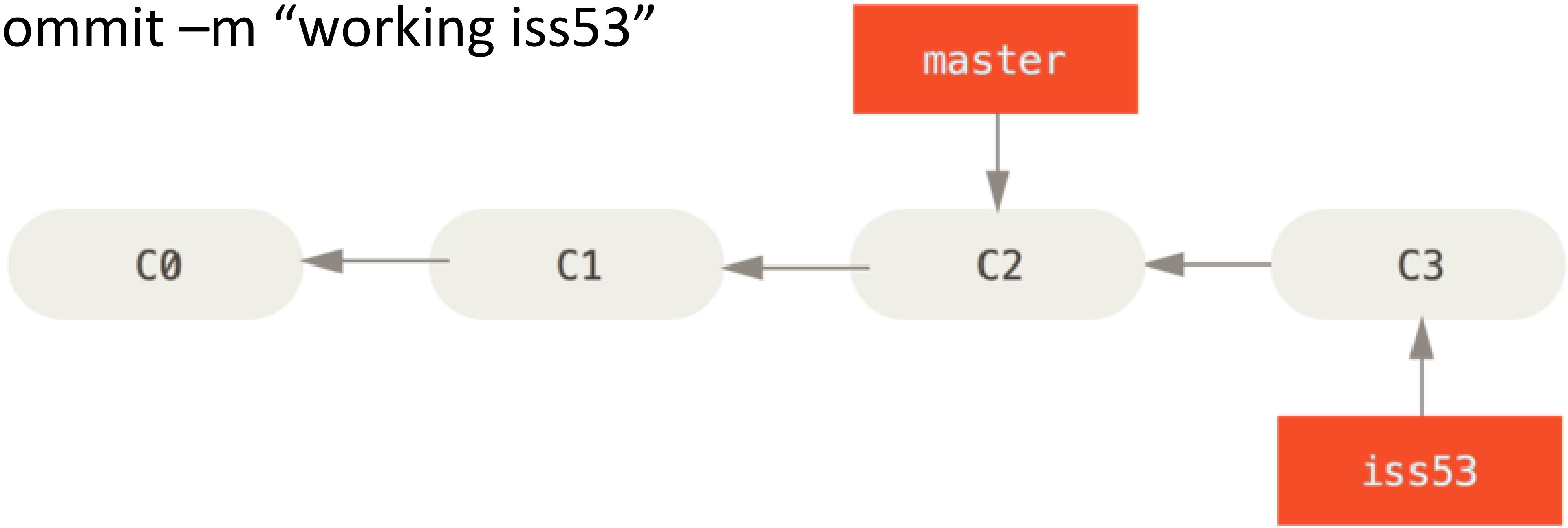
1. Create a branch

- git checkout –b iss53

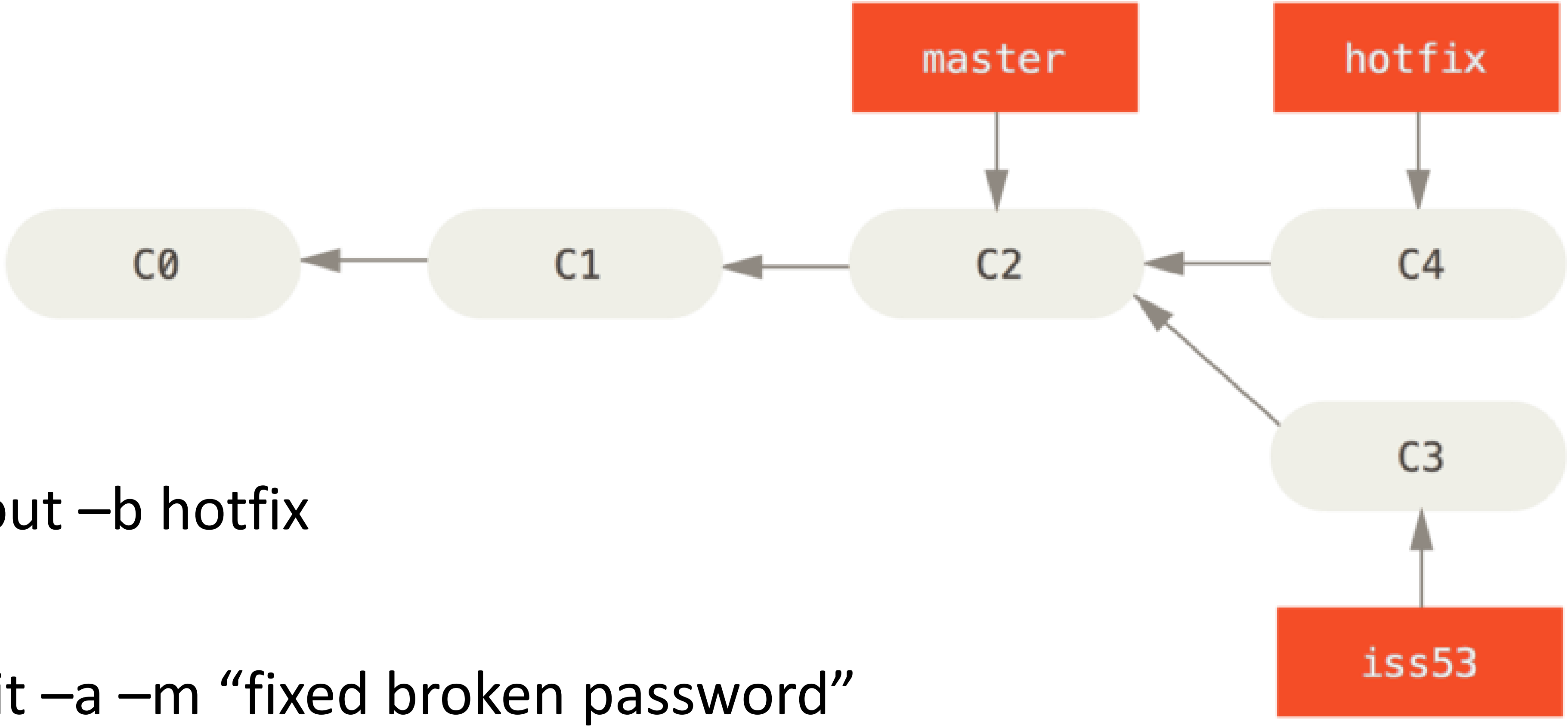


2. Add a commit on that branch

```
//change a file
git add file1
git commit –m “working iss53”
```



3. Need a new urgent branch for hotfix



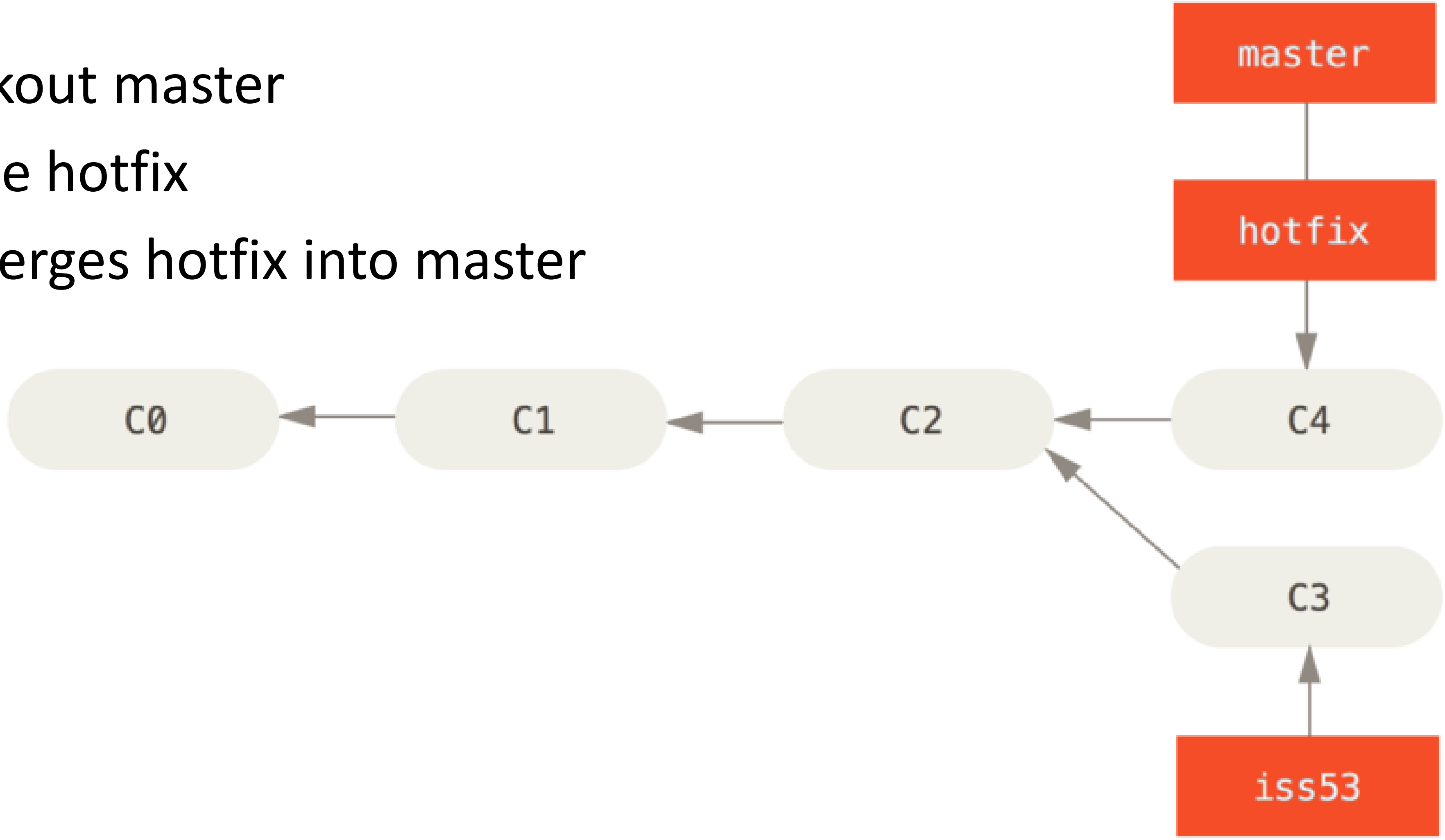
git checkout –b hotfix

//fix file

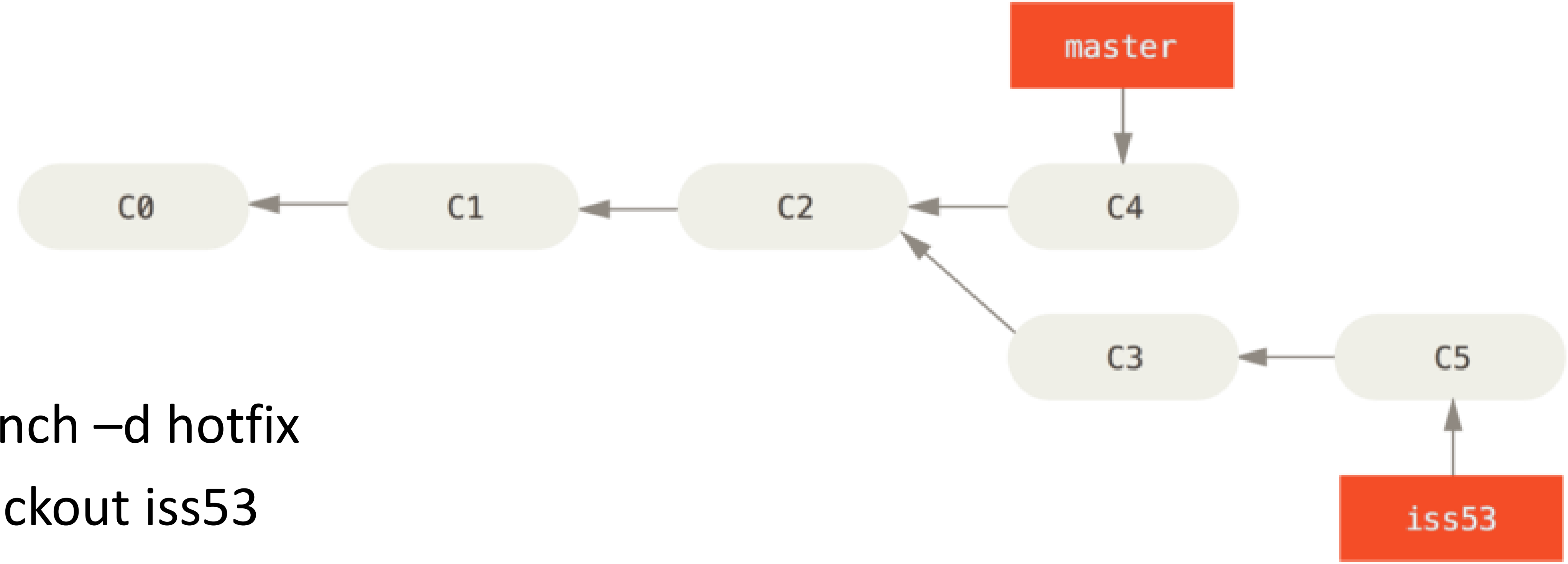
git commit –a –m “fixed broken password”

4. Complete Fast-Forward Merge

```
git checkout master
git merge hotfix
//this merges hotfix into master
```



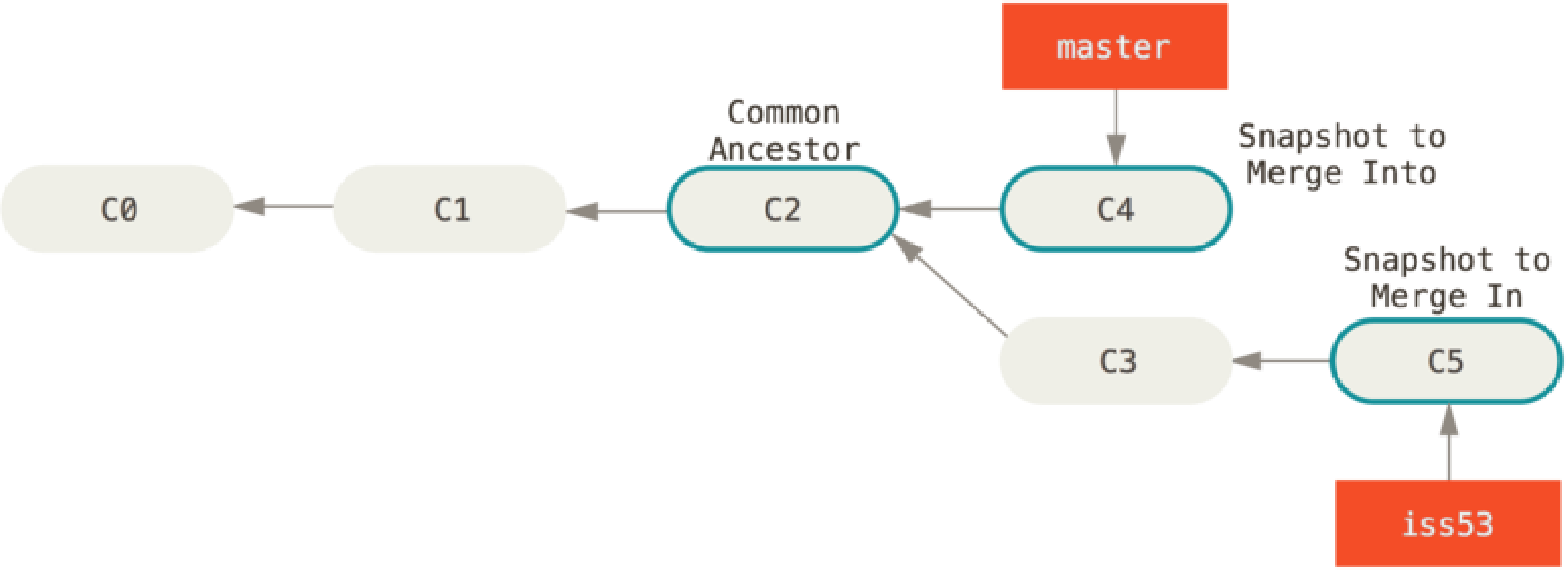
5. Go back to working on iss53



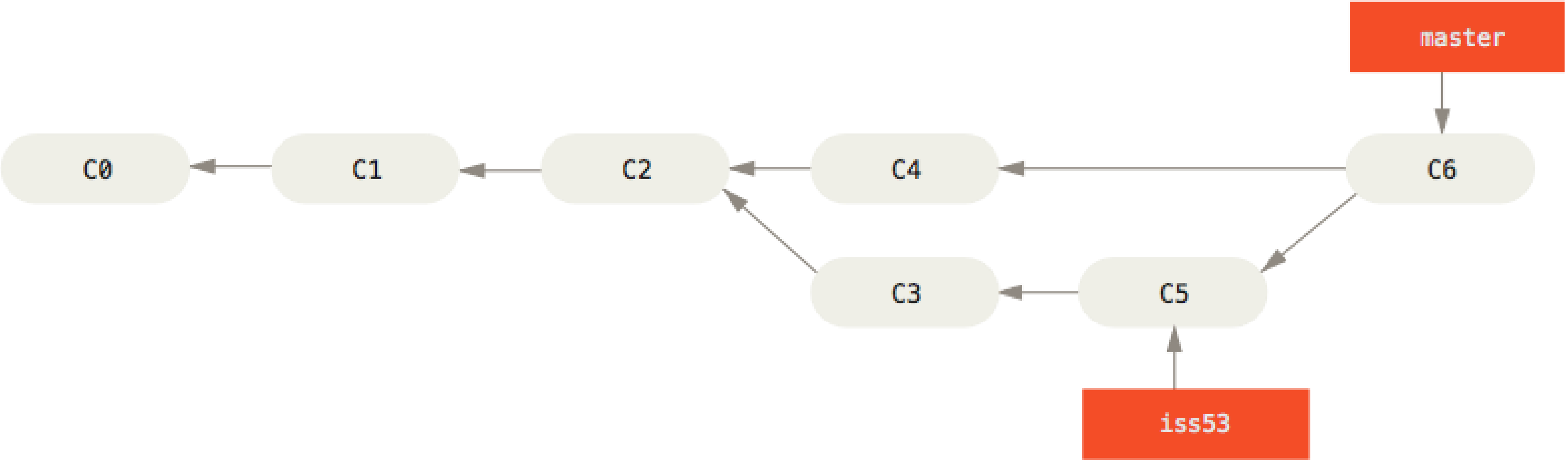
```
git branch –d hotfix
git checkout iss53
//make some changes
git commit –a –m “changed naming”
```

6. Merge iss53 into Master

```
git checkout master
git merge iss53
```



7. All merges completed



Git merge conflicts

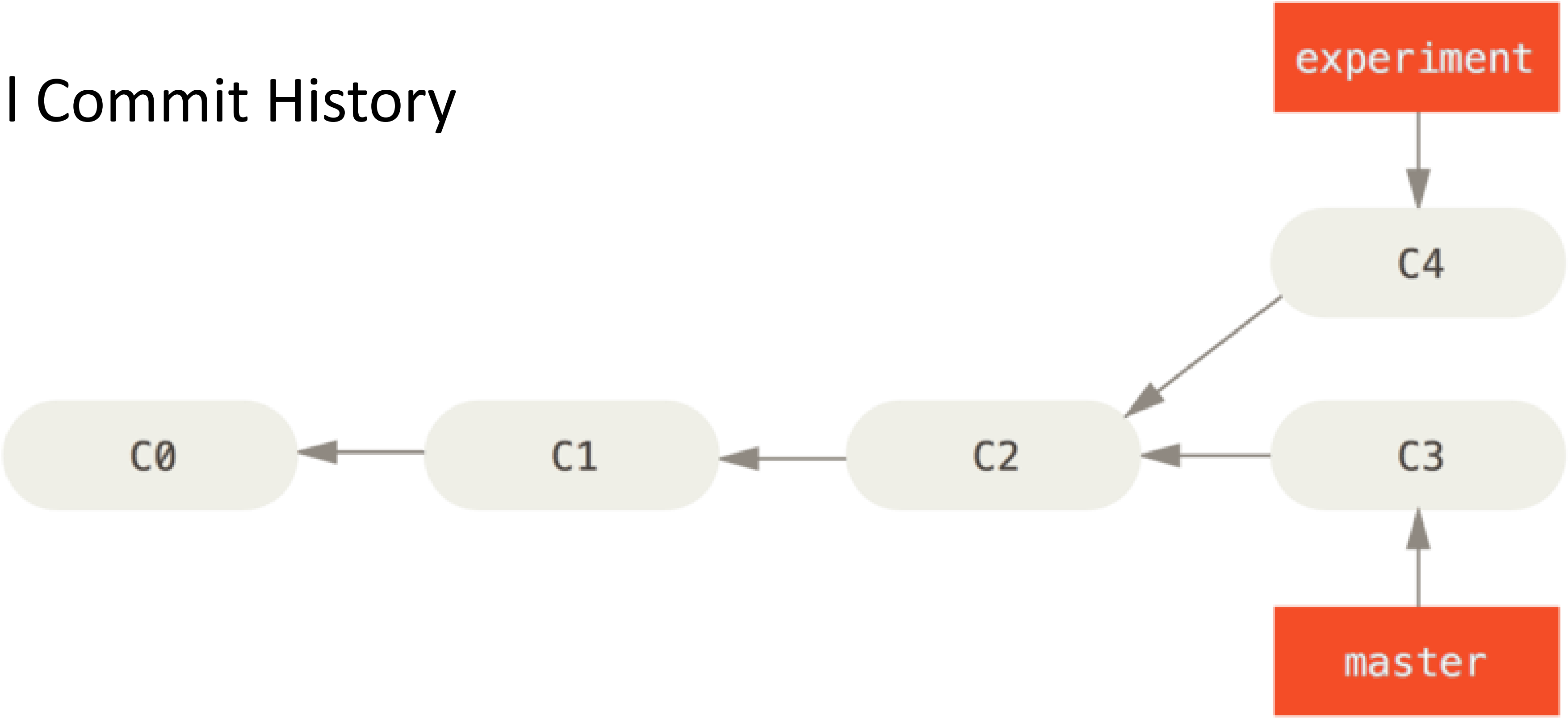
- Not every merge goes perfectly
- If hotfix and iss53 both modified the same part of the file, what is the final part we should keep? Git doesn’t know….
- In case of conflict, the merge will stop and tell you to resolve
  - Either go change the files manually
  - Use a visual tool like `git mergetool`

Merge vs Rebase

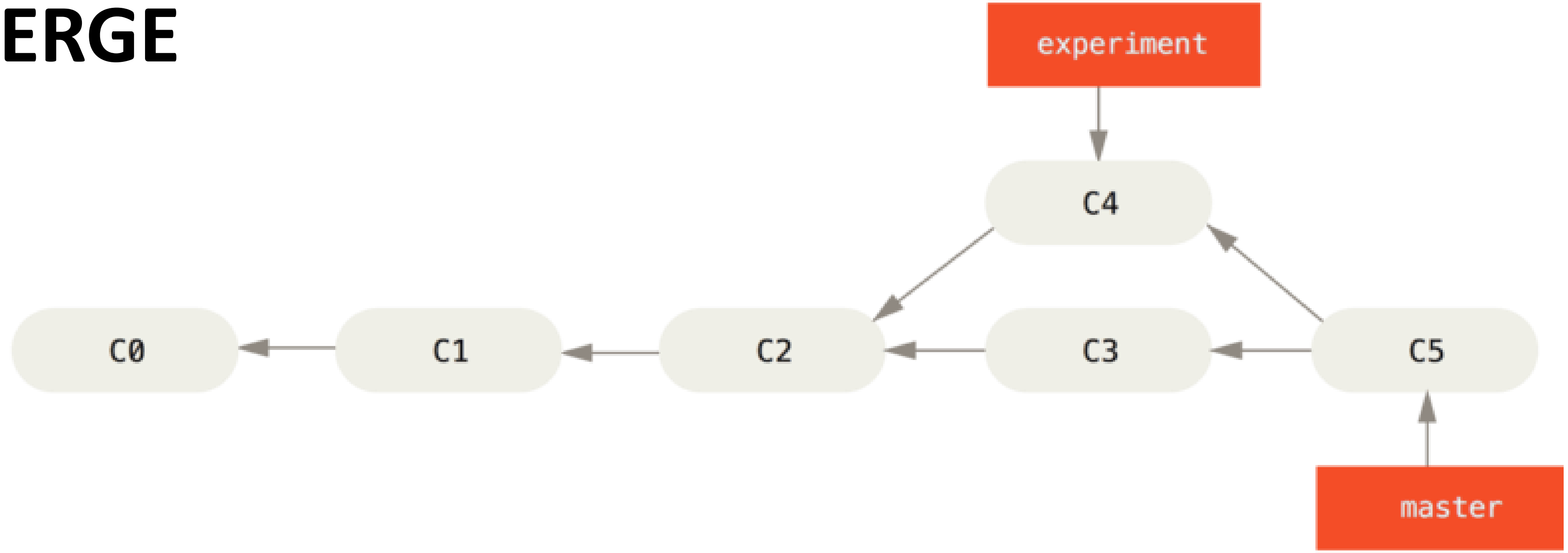
- Both designed to integrate two branches together
  - Merge takes the contents from the branch and tries to integrate into master
  - Rebase makes a new commit after master, and moves the branch to that
- Rebase will make for a “cleaner” more linear commit history, but does so by rewriting the history.
- Rebase addresses conflicts one at a time instead of all at once like merge
- General Tips
  - Rebase on local non-published work
  - Merge on shared, published code. So that way you don’t alter the commit history that other people may rely on.

Merge vs Rebase Example

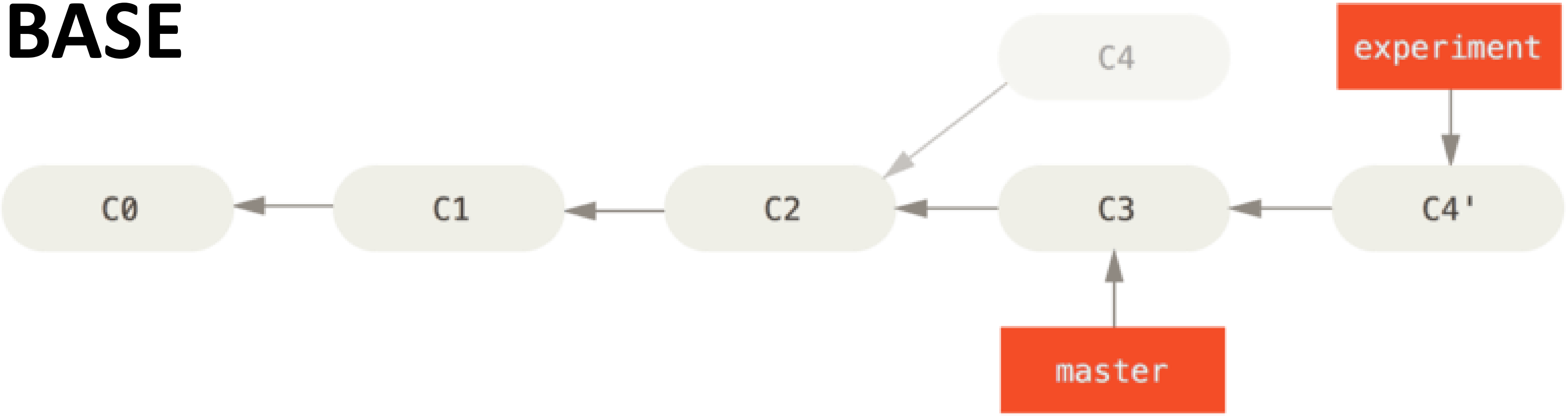
Original Commit History



MERGE



REBASE



Try at home - Git Workflow with Branches

- Create git repo on Github and clone to Server/computer
- To make a change, first create a branch and switch to it
  - git checkout –b <branchName>
- Work on feature incrementally and use Add/Commit
- When feature is ready, switch back to master and merge in
  - git checkout master
  - git merge <branchName>

Helpful Git Resources

- Git E-Book
  - https://git-scm.com/book/en/v2
- Many Youtube crash course videos for the basics, I like this one
  - https://www.youtube.com/watch?v=SWYqp7iY_Tc&t=1689s

Other Git Info

gitignore

- .gitignore file at the top of your working directory
- Specify any files that you want git to ignore by default
  - But you can forcibly add them if you want them.
- Example

```
meirovit$ cat .gitignore
#ignore .c files
*.c
```

git clean

- Removes all untracked file

git restore

- Restores files to their last commit state

git format-patch

```
git format-patch [numCommits] [CommitID] --stdout
```

- Used to generate a patch file relevant to that specific commit
- Examples
  - `git format-patch -1 <someCommitID> --stdout > patchFile`
  - `git format-patch -1 --stdout > patchFile`

Applying a patch generated by git

```
git am < patchFile
```

- You can use git to apply patches generated by ‘git format-patch’

Emacs Integration

Homework has a few different ways to use Emacs

- vc-revert (C-x v u)
    - Used to undo changes.
    - For lab 7 – think about which files were patched that you want to restore back
- Reverting selective hunks
    - Open version history (C-x v =)
    - Revert Hunk (C-u C-c C-a)