Department of Computer Science

CS411 Version Control With git

@perrydBUCS

What we're trying to solve

Gamut

- Track all changes to code base
- Rewindable history
- Collaboration
- Keep dev, test, prod code separate
- Work on features and fixes nondestructively

Two approaches

- **Centralized** repository (svn, cvs)
  - One copy of code base on server
  - Devs check out a file to work on locally
  - No one else can work on that file
  - When done, dev checks file in to the server
- Advantages:
  - Very clean separation of responsibility
  - Clean code history
- Disadvantages:
  - Only one dev can work on a file at a time (and vacations!)

- **Decentralized** (git)
  - Each dev has a copy of the code base
  - Concurrent work on a file is possible
  - Local versions (called branches) are embraced
- Advantages:
  - No locking of files
  - Concurrency
  - Simple branching
- Disadvantages:
  - Local branches must be merged
  - History can become complex

GitHub

- Serves as a central repository (repo) for a project
- Since git is distributed, the copy on GitHub is canonical *only by convention*
- When GitHub is the canonical version, a distributed workflow can be built on it for concurrent development
- While there are no 'official' workflows, a few models have emerged that are commonly used
- We'll focus only on one that is appropriate for your team projects

git concepts

- git records local changes made to files in a directory (and its subdirectories)
- Those records are essentially snapshots of the state of all files at a given moment
- Multiple concurrent histories, called **branches**, are used to isolate specific work, for example a bug fix or new feature
- Two branches can be **merged** together, combining all of the changes made to both branches
- Local copies can be synchronized with other developers' local copies, or with branches stored on GitHub

Commits: Saving changes to a file

- git only records changes when you tell it to, using the ‘add’ command
- ‘add’ is used to move the current state of a file into a staging area (it really should have been called ‘stage’ but wasn’t)
- Changes that have been staged are recorded with the ‘commit’ command
- The workflow is

edit -> stage (add) -> commit

Staging is a snapshot!

- Staging happens when `git add` is executed and *only* then
- If you stage a file, then make more edits, they will not be included in the next commit unless you `git add` them again

Your best friend: `git status`

- The `git status` command provides details of your current state and advice on what to do next

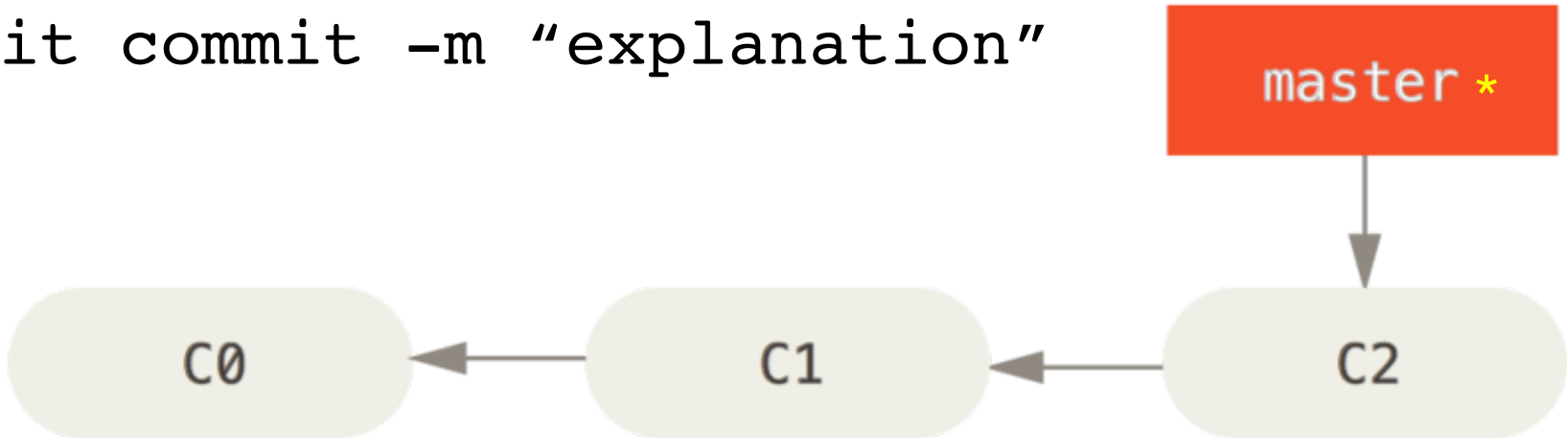
```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$
```

Branching and merging

- **Branches** are used to separate work from the main code base
- They let you work on new features, updates, bug fixes, and so on independent of the main code base
- In git it is common to create lots of new branches and delete them when you don't need them any more
- Work in a branch is rolled into the main line of code using the `git merge` command
- The following illustrations are from the 'official' manual at https://git-scm.com/book/en/v2

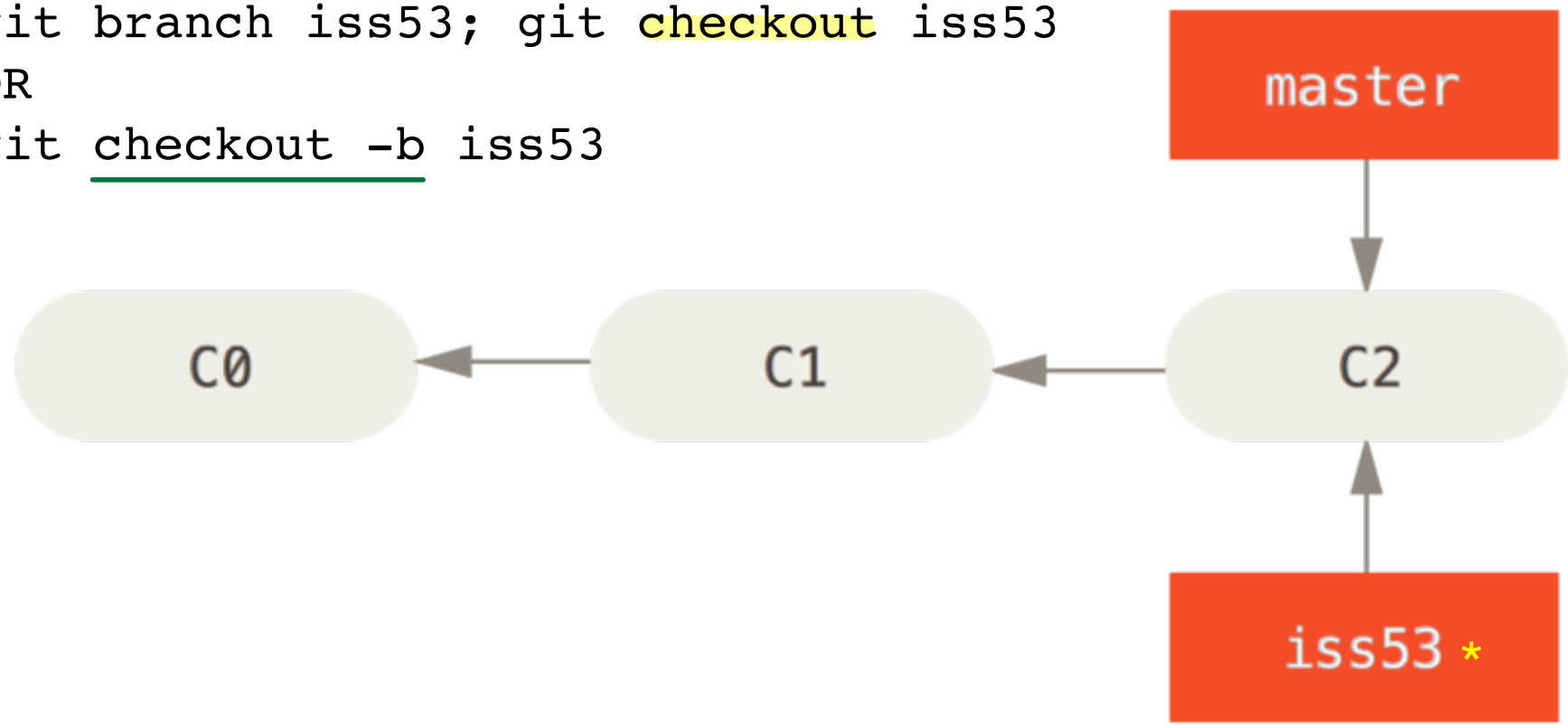
Work done on a single branch (master)

```
git commit -m “explanation”
```

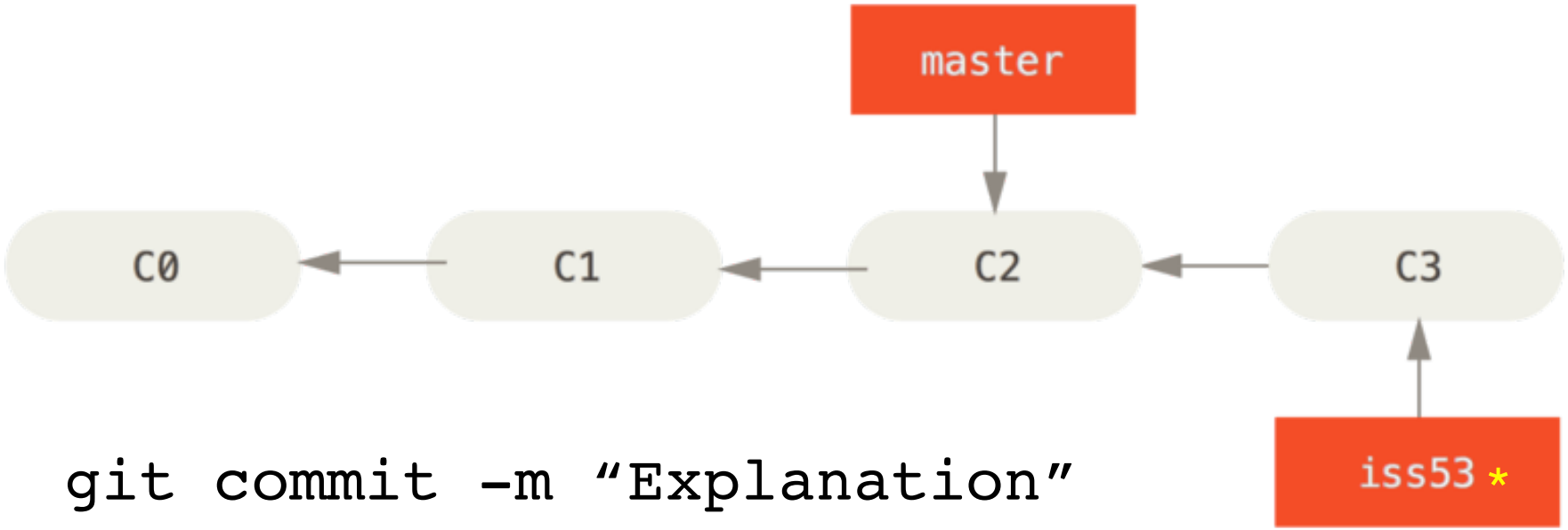


A new branch to work on issue 53

```
git branch iss53; git checkout iss53
OR
git checkout -b iss53
```

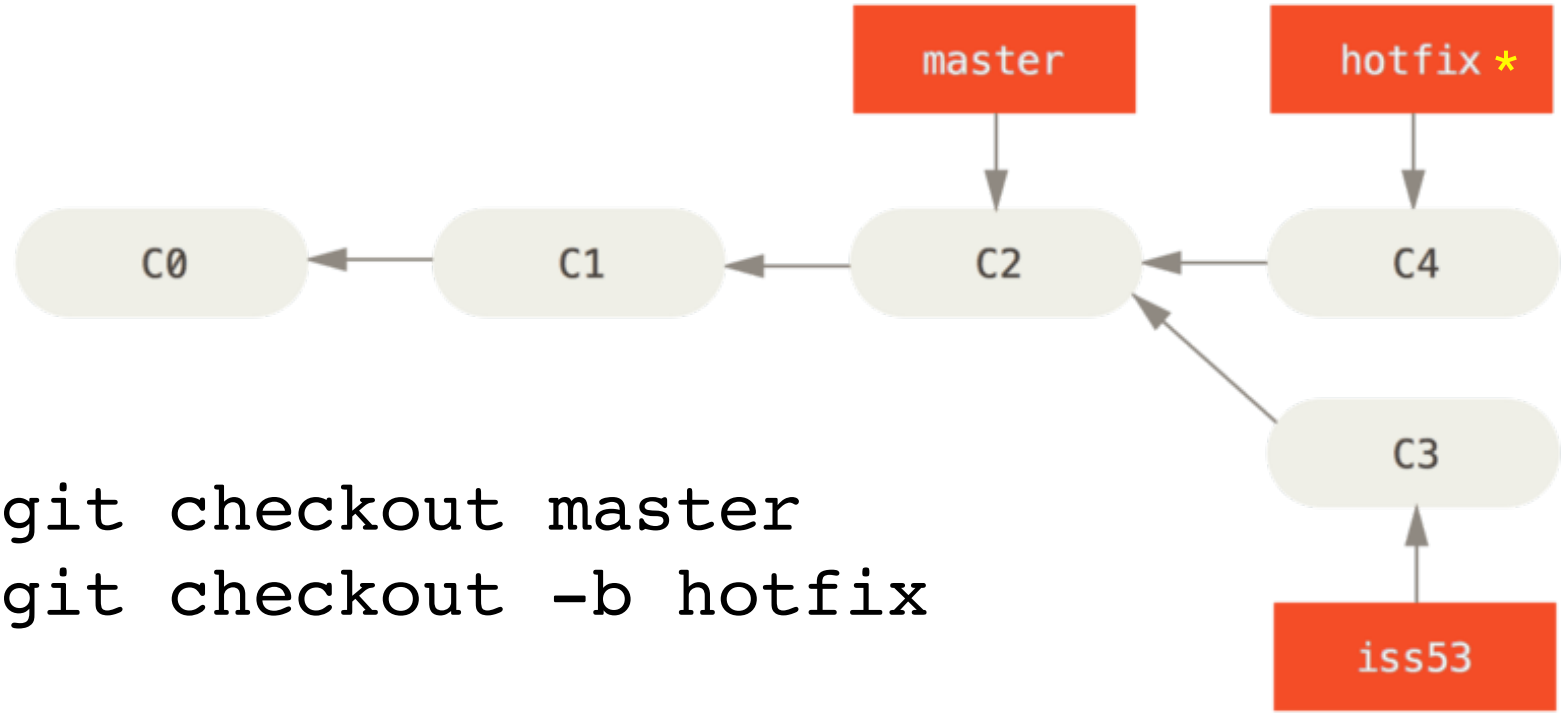


add and commit changes in branch iss53




```
git commit -m "Explanation"
```

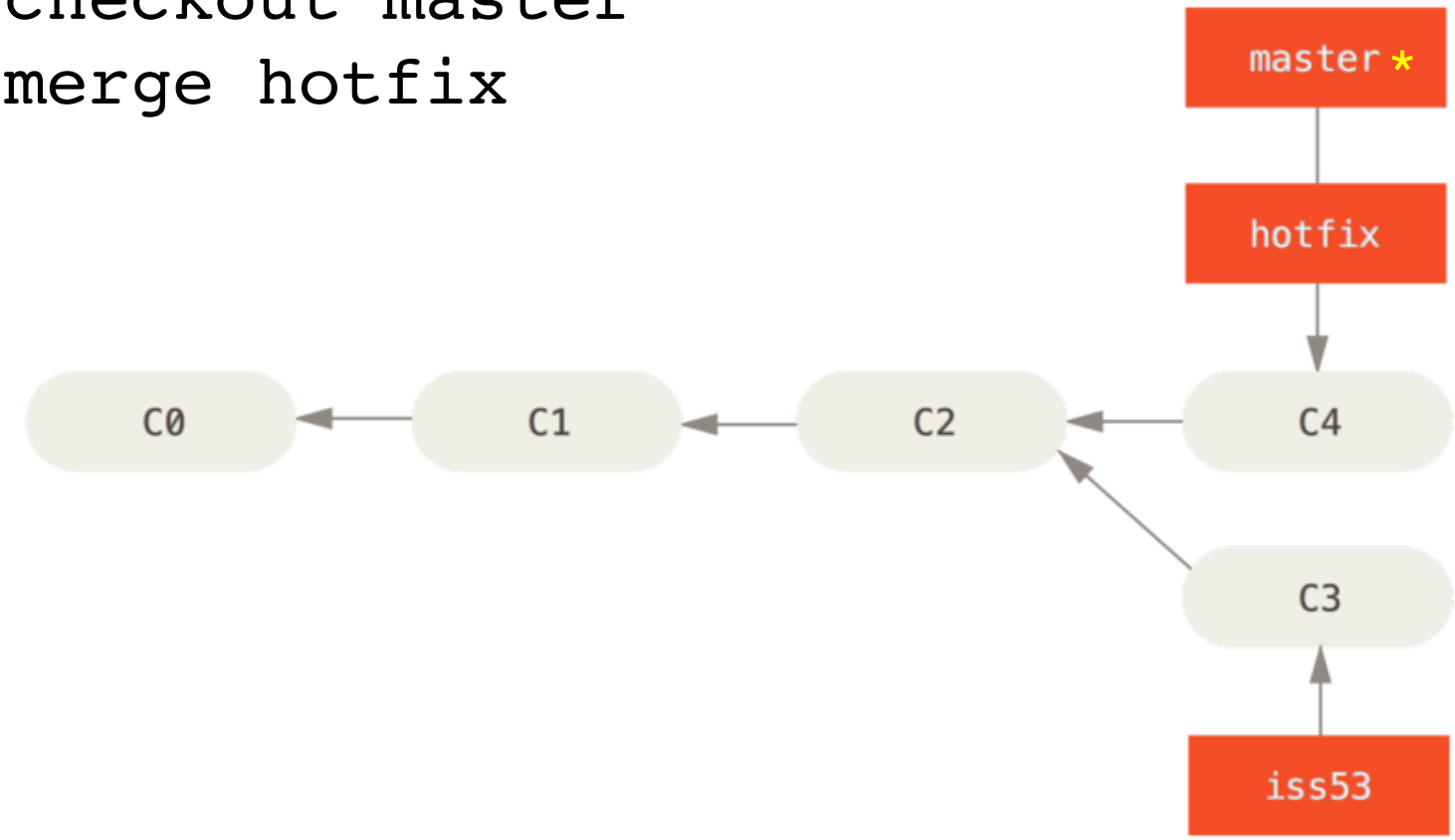
Move back to master, create new branch to work on a hotfix



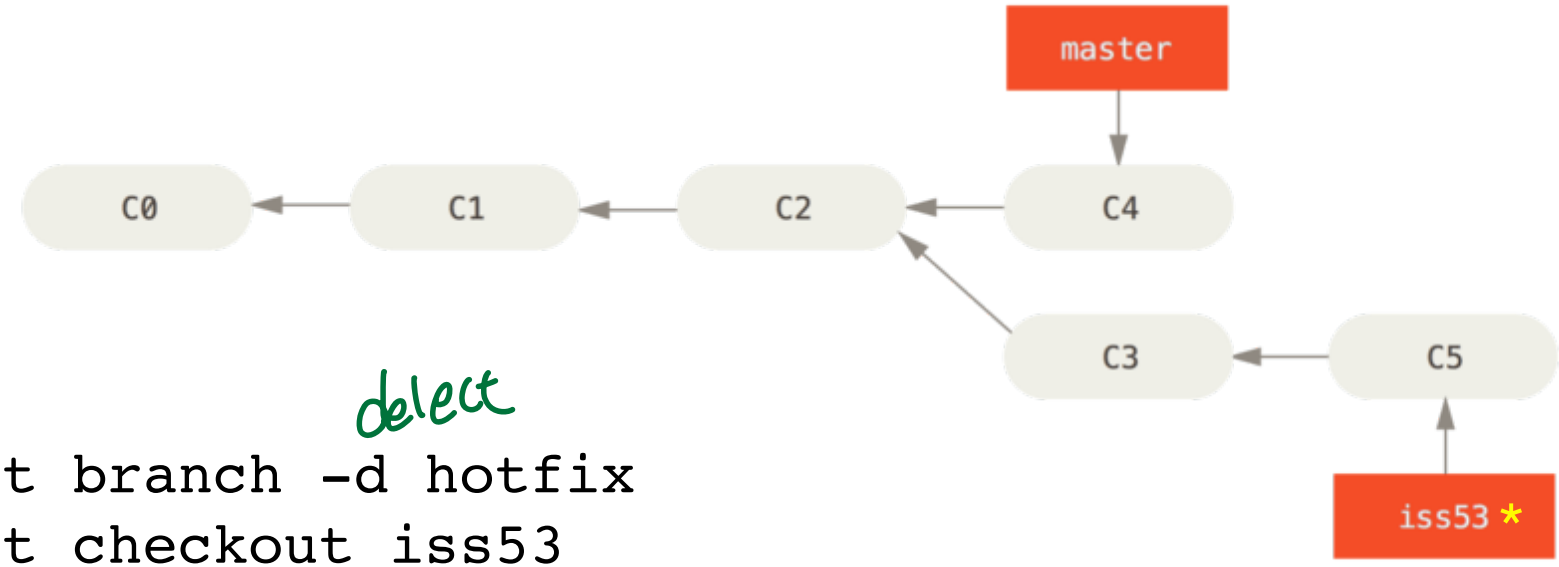

```
git checkout master
git checkout -b hotfix
```

hotfix is merged into master

```
git checkout master
git merge hotfix
```



New commit on iss53…hotfix is no longer needed and is deleted

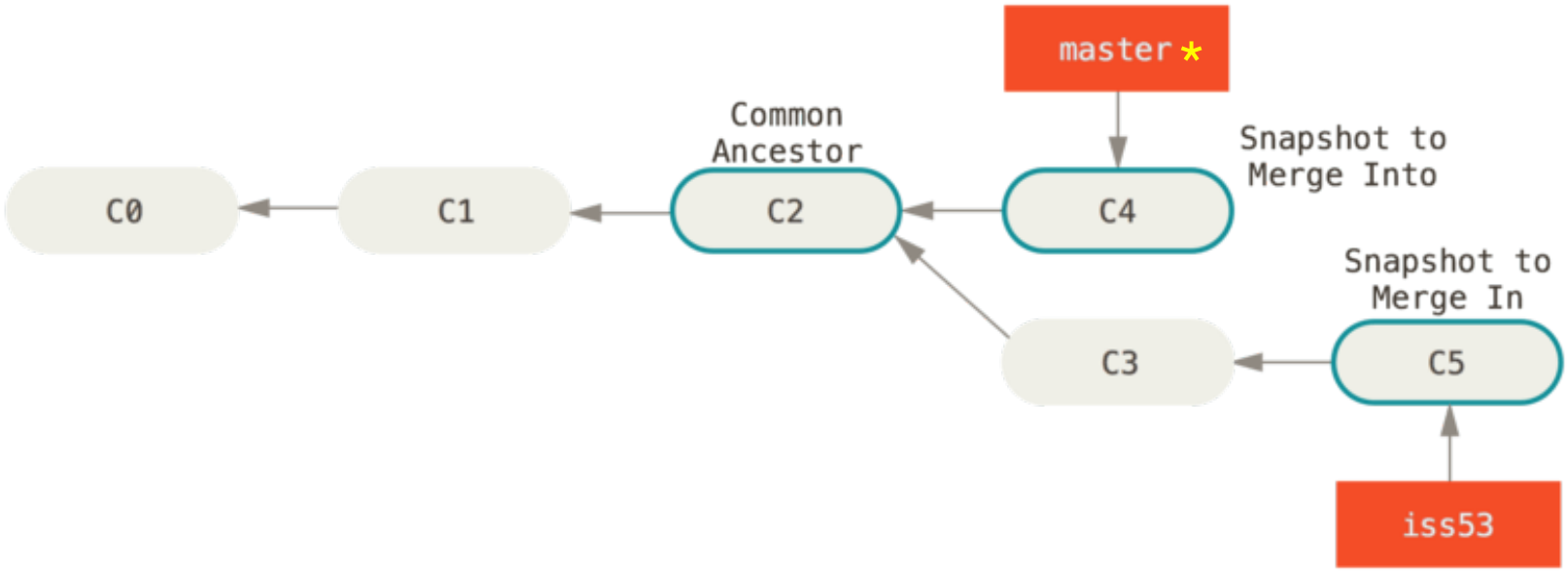


delect

```
git branch -d hotfix
git checkout iss53
<some work is staged with git add>
git commit -m “fixing iss53 prob”
```

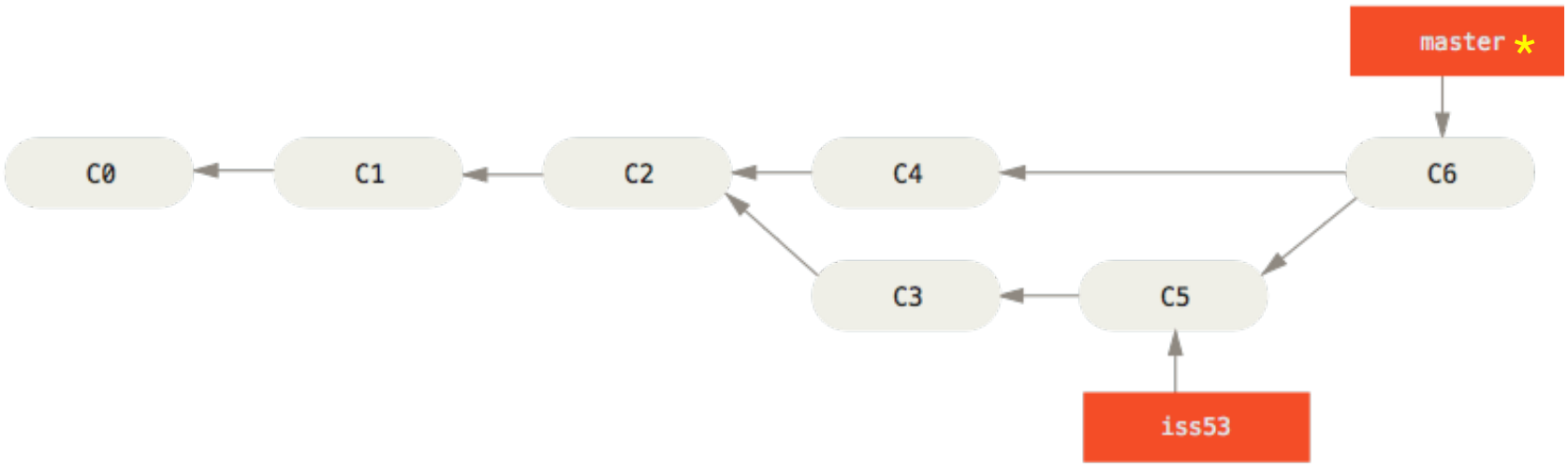
Getting ready to merge iss53 into master

```
git checkout master
```



Merge iss53 fixes into master

```
git merge iss53
```



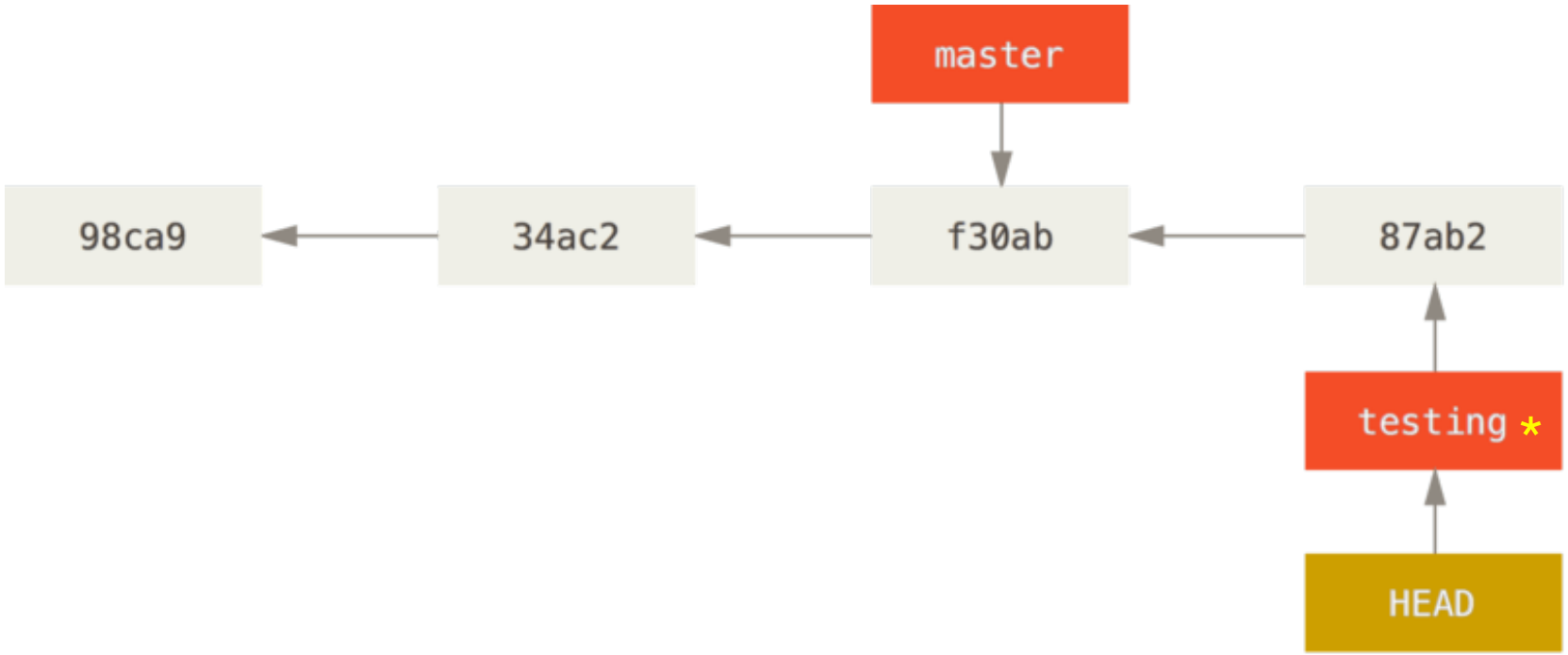
Basic flow

- Get up to date with the code you will be branching off of (often master but not always)
- Create a branch for a specific piece of functionality / bug fix
- Test your changes
- Merge your changes back into the main branch
- Delete the ‘feature’ branch

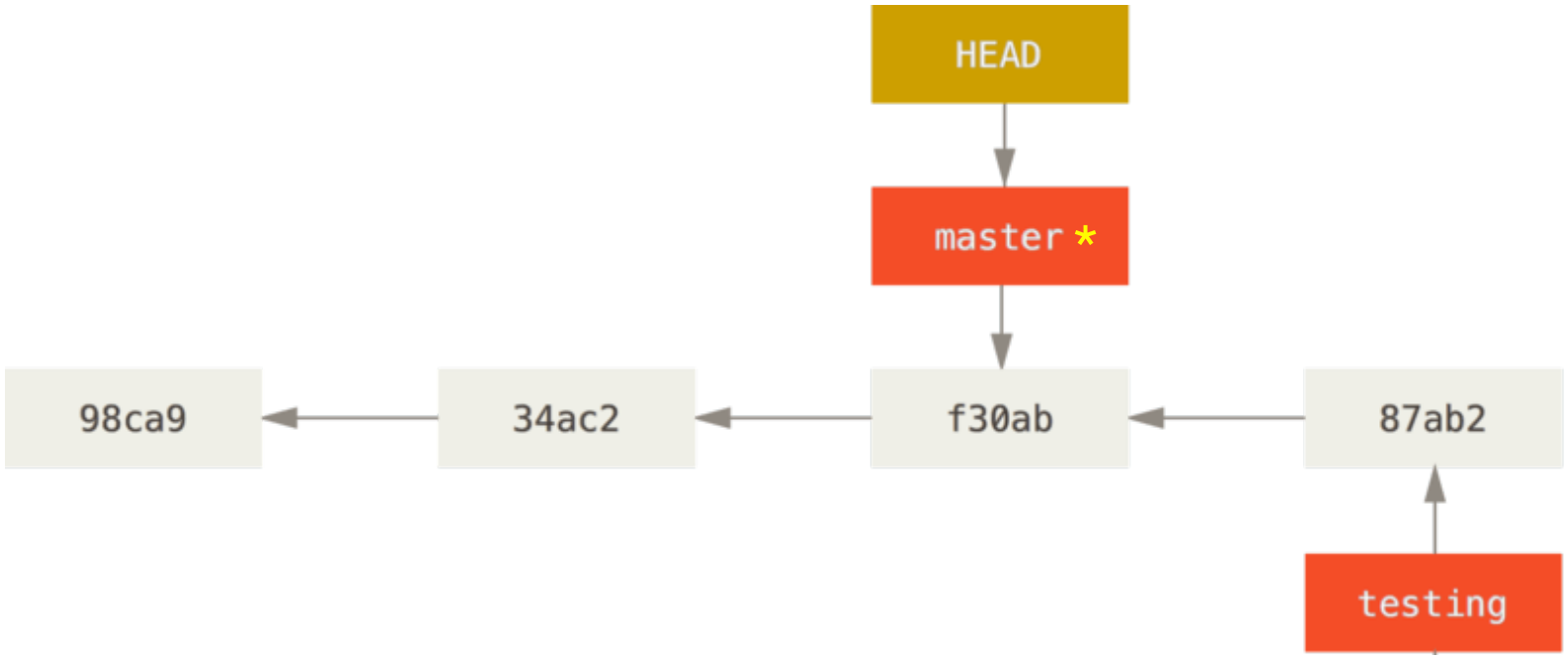
The HEAD pointer

- git keeps a pointer, called HEAD, that refers to the commit that you are currently working at (that's what the * was in the previous slides)
- Use `git checkout` to move HEAD around
- Normally we're moving to the tip of a branch, but you can also move to a specific commit if you need to
- When you issue `git branch`, the branch is created from wherever HEAD is pointing to

After issuing `git checkout testing`

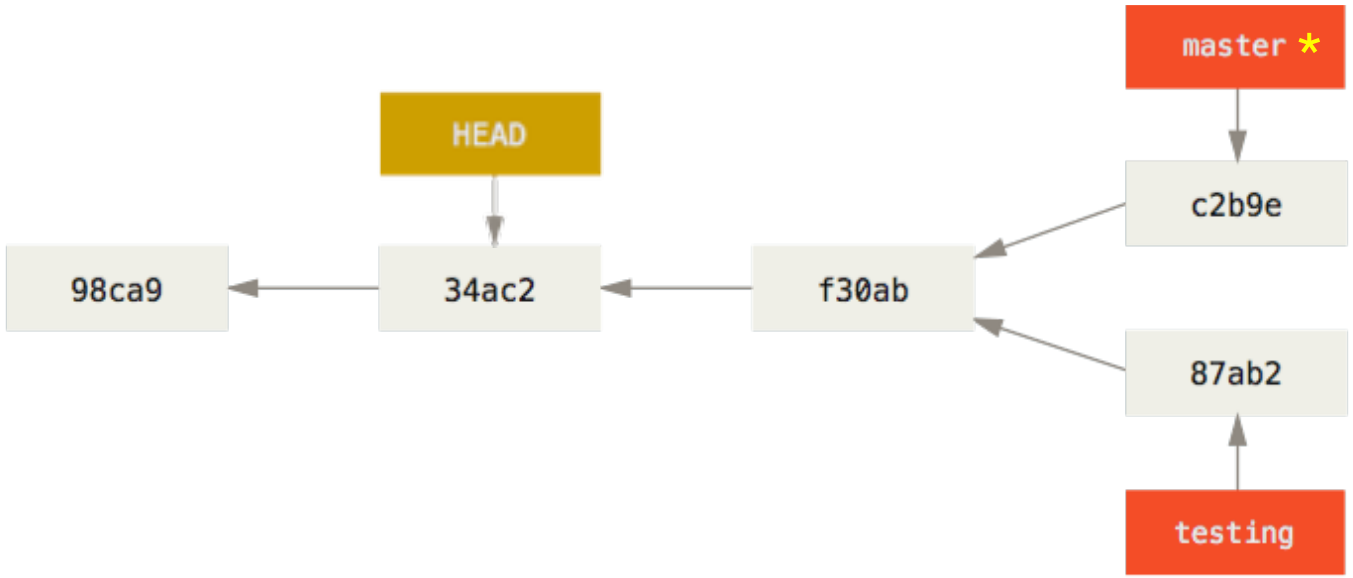


`git checkout master` moves back to the tip of master



Moving to a prior commit (this is called ‘detached HEAD’)

```
git checkout 34ac2
```



Stashing

- If you are working in a branch and have uncommitted changes, git will prevent you from switching
- This is because checking out a branch places all the files in your working directory in the state they were at when the branch was last committed
- That means that switching to a different branch when you have uncommitted changes in the current branch might overwrite those files
- To get around this, we use `git stash` to take a snapshot of those uncommitted changes

README.md has uncommitted changes

```
$ git commit -am "added Xs to README"
[quickTest 280414f] added Xs to README
 1 file changed, 1 insertion(+), 6 deletions(-)

$ vi README.md [make a change]

$ git checkout master
error: Your local changes to the following files would be overwritten by checkout:
    README.md
Please commit your changes or stash them before you switch branches.
Aborting

$
```

`git stash` saves the uncommitted changes to a stack

```
$ git stash
Saved working directory and index state WIP on quickTest: 280414f added Xs to README

$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with ‘origin/master'.

$ git stash list
stash@{0}: WIP on quickTest: 280414f added Xs to README
$
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
$
```

```
$
$ git checkout quickTest
Switched to branch 'quickTest'
$ git status
On branch quickTest
nothing to commit, working tree clean
$ head README.md
xxxx on quickTest

$ git stash list
stash@{0}: WIP on quickTest: 280414f added Xs to README

$ git stash pop   //or apply, which leaves stash on the stack
On branch quickTest
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
$ git add README.md
$ git commit -m "Forgot trailing Xs"
[quickTest 9bb1548] Forgot trailing Xs
 1 file changed, 1 insertion(+), 1 deletion(-)

$head README.md
xxxx on quickTest xxx
```

Conflicts

- Since git is distributed, it is possible (even likely) that two devs will work on the same file in different branches
- If the changes on the file conflict with each other, the conflict must be resolved
- This is usually a manual process
- The merge will pause to give you a chance to figure out which change to keep
- Once you are done, the merge resumes

Tools for managing conflicts

- A merge conflict creates a new file that marks the conflicting chunks
- You can open it with a text editor and resolve the conflict there, however it can get messy
- Most folks use a tool like mergetool (installed on MacOS when you install XCode) or gitKraken or others
  - These tools give you a side-by-side view
  - They let you click-and-pick which part of the code to use or to drop

Both master and quickTest have Xs in line 1 but done differently…which is correct?

```
$ git stash drop
Dropped refs/stash@{0} (f2d78468f616bde989d34d7e322de7ce6d8d2b9f)
$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ emacs README.md. [make a change]

$ git commit -am "Added Xs in master"
[master b93e76b] Added Xs in master
 1 file changed, 1 insertion(+), 6 deletions(-)
$
$ git merge quickTest
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
$
```

README.md now has info about the conflict

```
$less README.md

<<<<<<< HEAD
*** These are in master ***
=======
xxxx on quickTest xxx
>>>>>>> quickTest
# `angular-seed` — the seed for AngularJS apps

This project is an application skeleton for a typical [AngularJS][angularjs] web
app. You can use it…
```

Fix the conflict (either manually or with a visual tool) and commit to complete the merge

```
$ emacs README.md [resolve conflicting code]

$ git commit -am "Fixed merge conflict in README, chose quickFix text"
[master a0e7b2b] Fixed merge conflict in README, chose quickFix text

$ git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
$
```

Project workflow with gitHub

- There are two main ways to set up a repo on github
  - Push existing local files to a new repo
  - Set up a new repo on github and clone it
- We'll look at the second method, which is the simpler of the two
- For the first method, a good tutorial is at https://www.digitalocean.com/community/tutorials/how-to-use-git-effectively

Project workflow: Setup for LEAD

- All members create gitHub account if it doesn't exist
- One team member (LEAD) creates a repo
- LEAD adds MEMBERs to collaborators list (add collabs in the Settings page (gear icon))
- MEMBERs accept email invite to be collaborator
- MEMBER navigates to github repo, click on green 'Clone or download' button, copy URL displayed
- MEMBER: From a terminal on your local machine, move to the directory you want your local repo to be
- MEMBER: git clone <URL you copied earlier>

Project workflow: Setup

- MEMBER creates a personal branch (i.e. perryd would do `git checkout -b perry`)
- MEMBER pushes personal branch to set up tracking (`git push --set-upstream origin perryd`)

Project workflow: Doing work

- MEMBER: move to project directory on your machine
- Switch to your personal branch

```
git checkout perryd
```

- Update with any changes made since last time you were working

```
git pull origin master
```

- Create a new topic / feature branch to do work on a specific item

```
git checkout -b oauth
```

Project workflow: Doing work

- After completing work on the topic branch, merge it into your personal branch

```
git checkout perryd
git merge oauth
```

- Push your personal branch to the project's gitHub repo

```
git push
```

- Notify LEAD that your changes are ready to merge into the release branch with a pull request
  - Log onto gitHub, navigate to project repo, click on New pull request
  - base: master <- compare: <your personal branch>
- LEAD evaluates request, requests comments, merges into masterSave files on local branch to remote repo

Commands used in demo

- git init //create a new local repo (from current directory)
- git add . //add any existing files to local repo
- git commit -m “Message” file //commit local changes
- git remote add origin URL //connect to remote repo
- git remote -v //show remote repo connections
- git branch //display all branches
- git pull //fetch remote files

Links and tools

- https://datasift.github.io/gitflow/IntroducingGitFlow.html
- gitKraken: https://www.gitkraken.com
- Official git docs (the 'Book' is pretty good): https://git-scm.com/doc
- git interactive playground / challenges: https://learngitbranching.js.org/?NODEMO. (type 'levels' at the prompt to start game)