COSC 340: Software Engineering

Version Control with Git

Audris Mockus

Notes adapted from:

Michael Jantz

Pro Git, 2nd Edition by Chacon and Straub

Available online at: https://git-scm.com/book/en/v2/

What is Version Control?

- A system that records changes to a file or set of files over time so that you can recall specific versions later
  - Often used to write software
  - Useful for any collaborative document / project
- Version control systems (VCS) can:
  - Revert specific files to a previous state
  - Revert the entire project to a previous state
  - Compare changes over time
  - See who introduced an issue and when
  - **Reproduce *EVERY* state**

Types of Version Control

- Local Version Control
  - Backup files
    - VMS from VAX minicomputers
  - tar, diff, patch
  - Example: *rcs, sccs*
    - *co – chekout*
    - *ci - checkin*
    - *rcslog – see history*

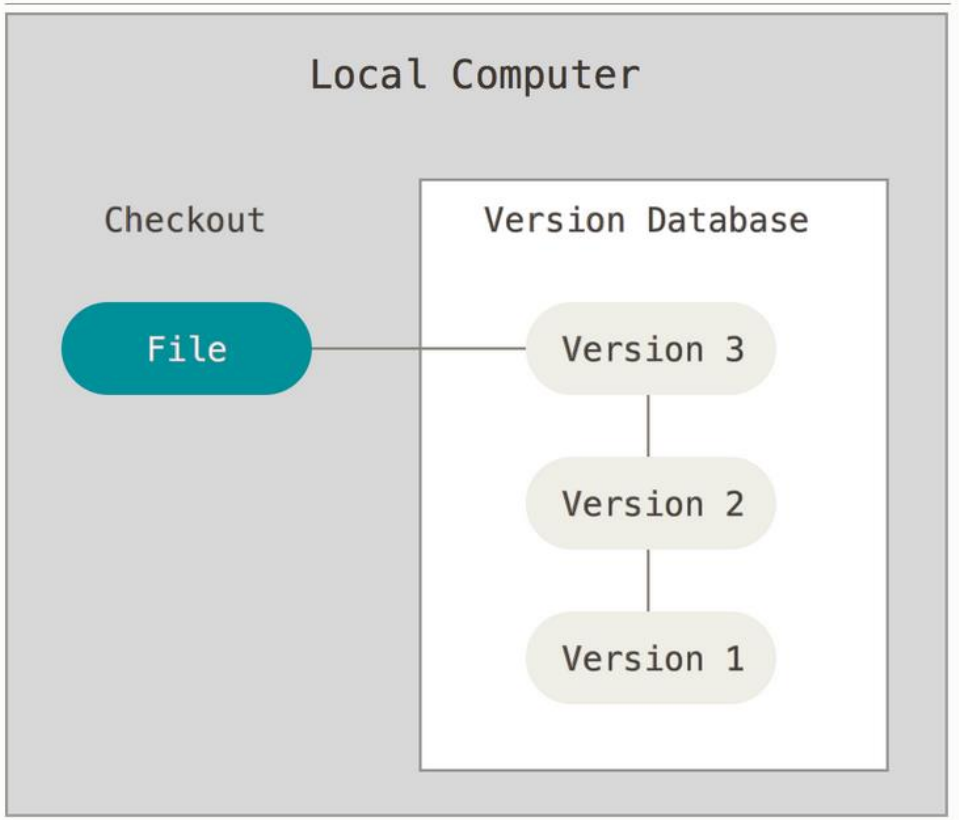
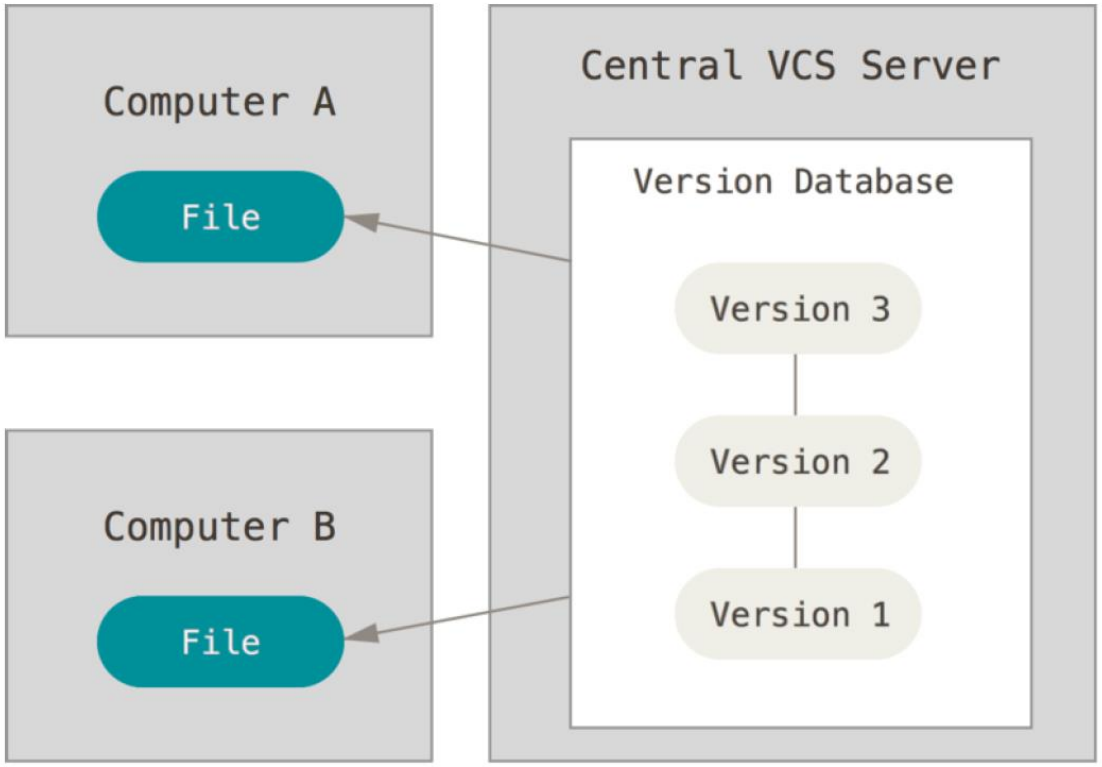


Figure 1-1. Local version control.

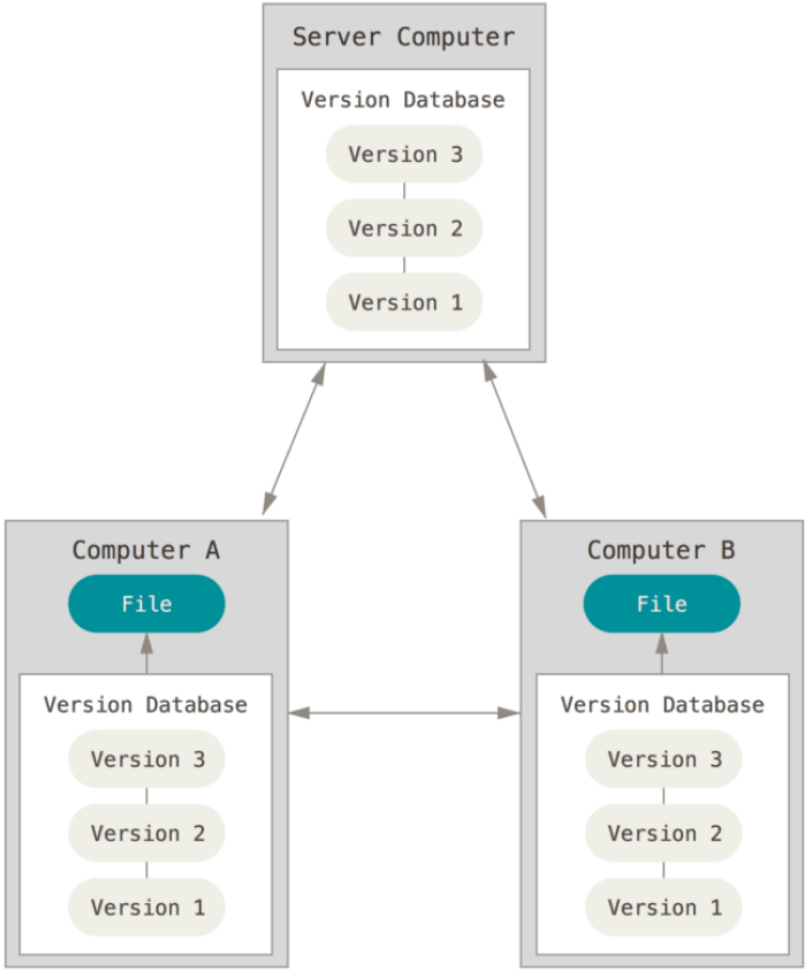
Types of Version Control

- Centralized VCS
  - Enables collaboration with developers on other systems
  - Single server contains all versioned files, clients check files in and out from the central repository
  - Risks from keeping all files in one central location
  - Examples: CVS, Subversion, Perforce



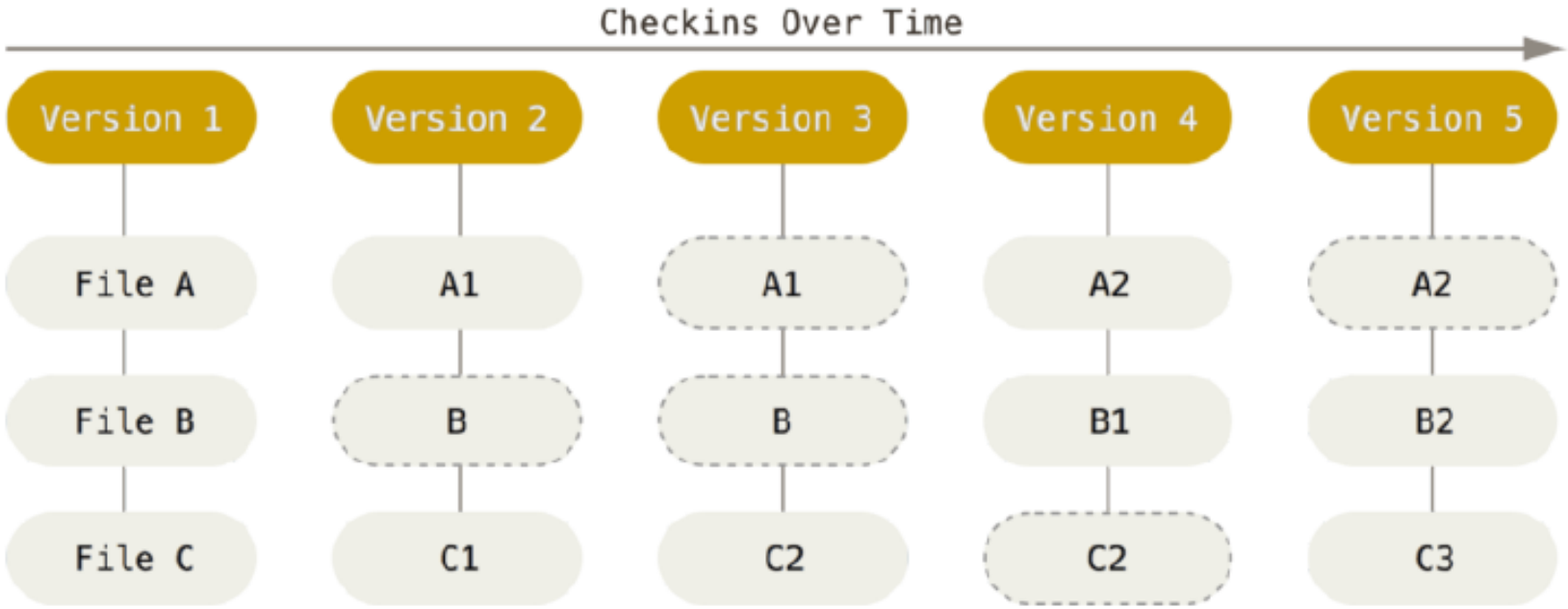
Types of Version Control

- Distributed VCS
  - Clients fully mirror the repository
  - Every clone is a full back-up of the data
  - Examples: Git, Mercurial, Bazaar, Darcs



Git Basics

- Git is actually a content tracking not version control: each commit represents a full filesystem



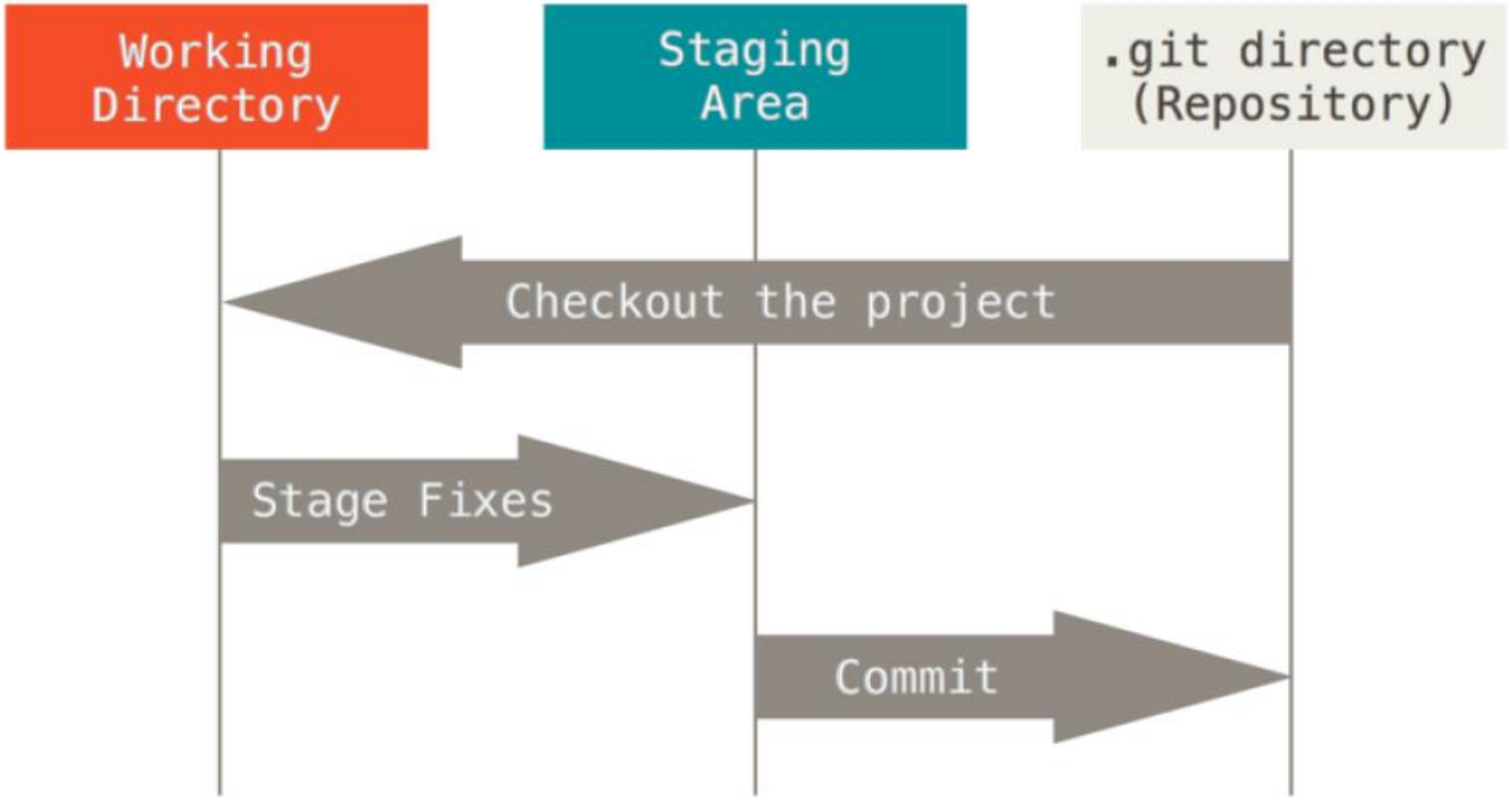
Git Basics

- Nearly every operation is local
  - Fast and easy to look up and compare files from the past
  - Can work offline
- Git has integrity guarantees
  - Everything is check-summed
  - Check-summing is done by computing sha-1 hash based on contents of a file or directory structure
- Git generally only adds data
  - No danger of really screwing things up

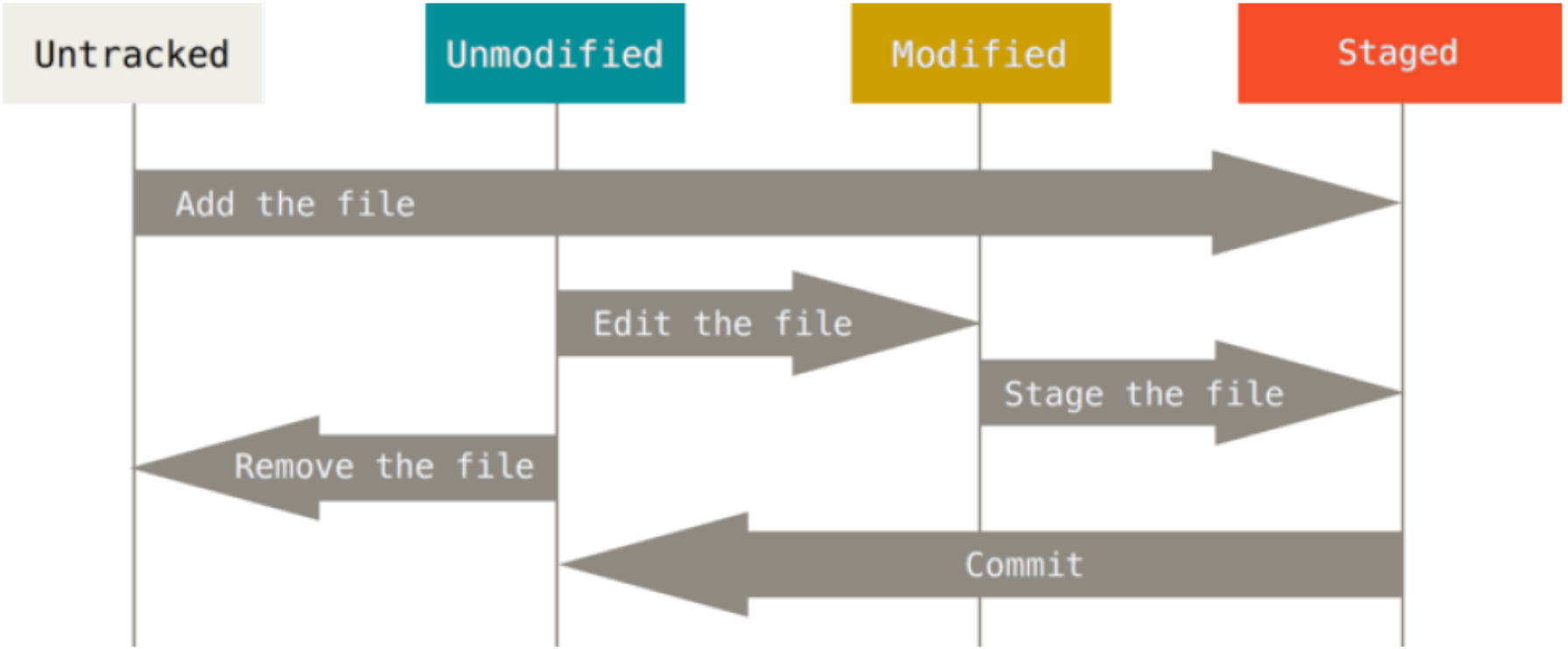
Three States for Files in Git

- Committed
  - Data is safely stored in your local database
- Modified
  - File has been changed, but not committed to your database
- Staged
  - A modified file is marked to go into your next commit snapshot

Three Main Sections of a Git Project



Tracked vs. Untracked Files



Creating pull request

- Fork and clone students repository
  - Fork using GUI or GH API https://github.com/COSCS340/students
  - git clone https://github.com/youtghid/students
- Add a yourutkid.md file, commit, and push:
  - cd students
  - Edit yourutkid.md
  - git add yourutkid.md
  - git commit –m "added my interests"
  - git push –u origin master
- Create and submit pull request
  - Using GH GUI or API

Working with Files

- *add* stages a file or directory (directories are added recursively)
  - git add file.txt
- *status* tells you the status of files in the repo
  - git status
  - git status –s (short version)
- *diff* compares files
  - git diff (compares working directory with staging area)
  - git diff --staged (compares staged changes to last commit)
  - git diff --cached (same as git diff --staged)

Working with Files

- *commit* creates a new revision with your staged changes
  - git commit (will open a text editor for you to document your commit)
  - git commit –m "document string" (to avoid opening an editor)
  - git commit –v (displays differences of what you're committing)
  - git commit –a (automatically stage every file that is tracked and then commit)
- *rm* stages a removal of a file
  - git rm file.txt
  - git rm --cached file.txt (removes a file from the staging area)
- *mv* renames a file
  - git mv file.txt new_file.txt

The .gitignore File

- Tells git that some classes of files should not be automatically added or even shown as being untracked.
- Lists filename patterns that should be ignored
- Placed in the directory in which you want the rules to apply (rules are applied recursively to all subdirectories)
- List of useful .gitignore files here:
  - https://github.com/github/gitignore

Viewing the Commit History

- *log* shows commit history
  - git log
  - git log -p (shows differences in each commit)
  - git log -p -2 (shows differences of only the last two versions)
  - git log --pretty=oneline (easy-to-read one line format)
  - git log --pretty=format:"…" (allows you to specify your own format string)
  - git log --since=2.weeks (show only commits in the last 2 weeks)
  - git log --author="Audris Mockus" (show only commits by that autor)
  - git log -Sstring (show only commits that added or removed the string 'string')
  - git log -- file.txt (show only commits that modified file.txt)

Undoing Things

- To add to a previous commit, use --amend:
  - git commit –m "initial commit"
  - git add forgotten_file.txt
  - git commit --amend
- To unstage a staged file, use *reset*:
  - git reset HEAD file.txt
- To unmodify a modified file, use *checkout*:
  - git checkout -- file.txt

Working with Remote Repositories

- *remote* shows your remote repositories
  - git remote
- *fetch* gets data from your remote repository
  - git fetch [remote-name] (leave remote-name blank to fetch from origin)
- *push* pushes data to the remote repository
  - git push [remote-name] [branch-name]
  - git push origin master (most common)

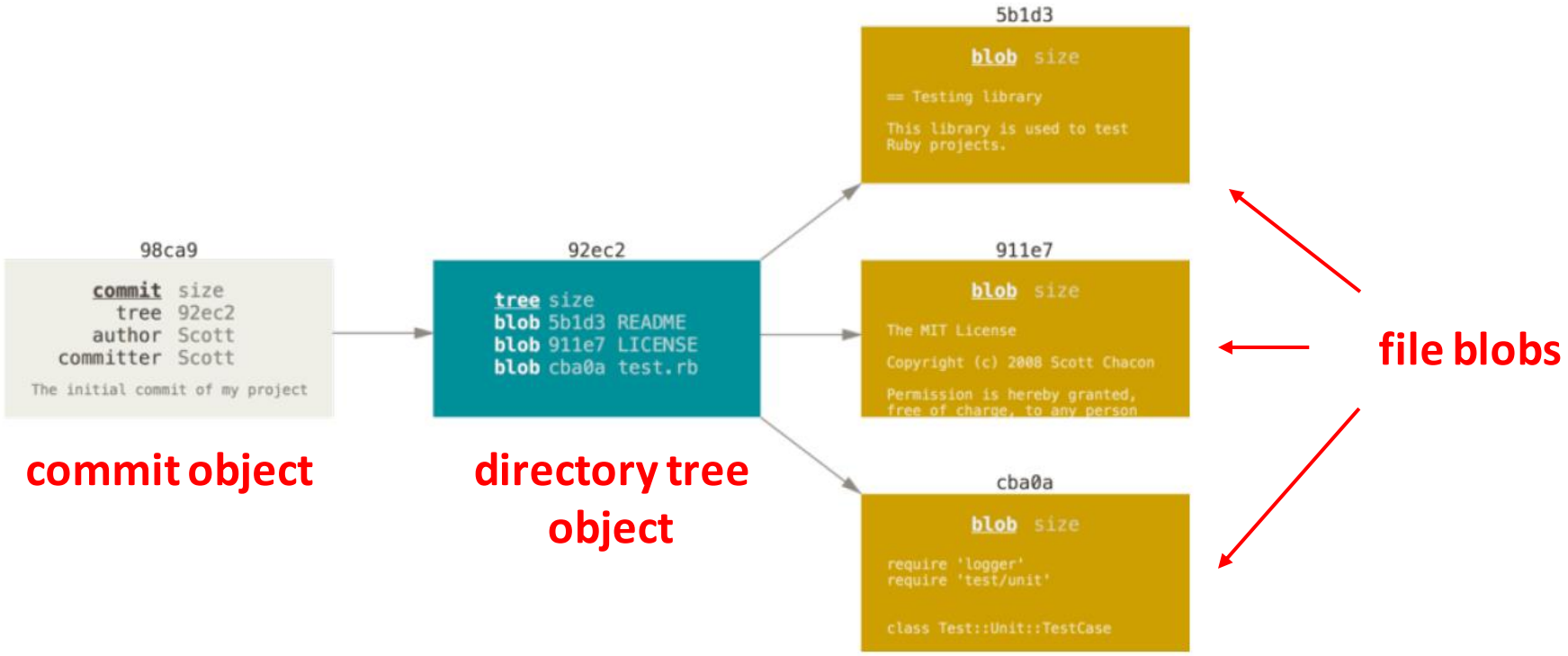
Branching in Git

- Branching means to diverge from the main line of development
  - Allows you to continue work without messing with the main line
- Git branching is lightweight
  - Does not copy entire source tree
  - Encourages workflows that branch and merge often

Commit Objects

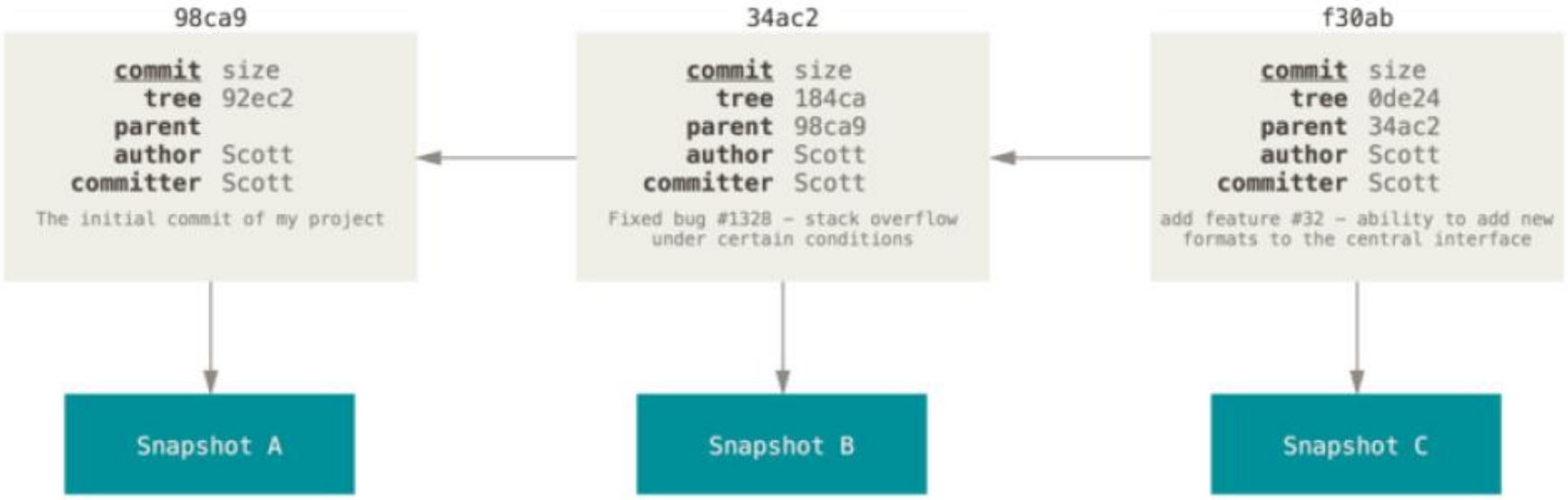
- A commit object that contains a pointer to the snapshot of the content you stored. The commit object includes:
  - Author name and email
  - Message attached to the commit
  - Pointers to the commit(s) that came directly before it (its parents)
    - Zero parents for the initial commit, 1 parent for a normal commit, multiple parents for a merge of two or more branches

Commit Objects



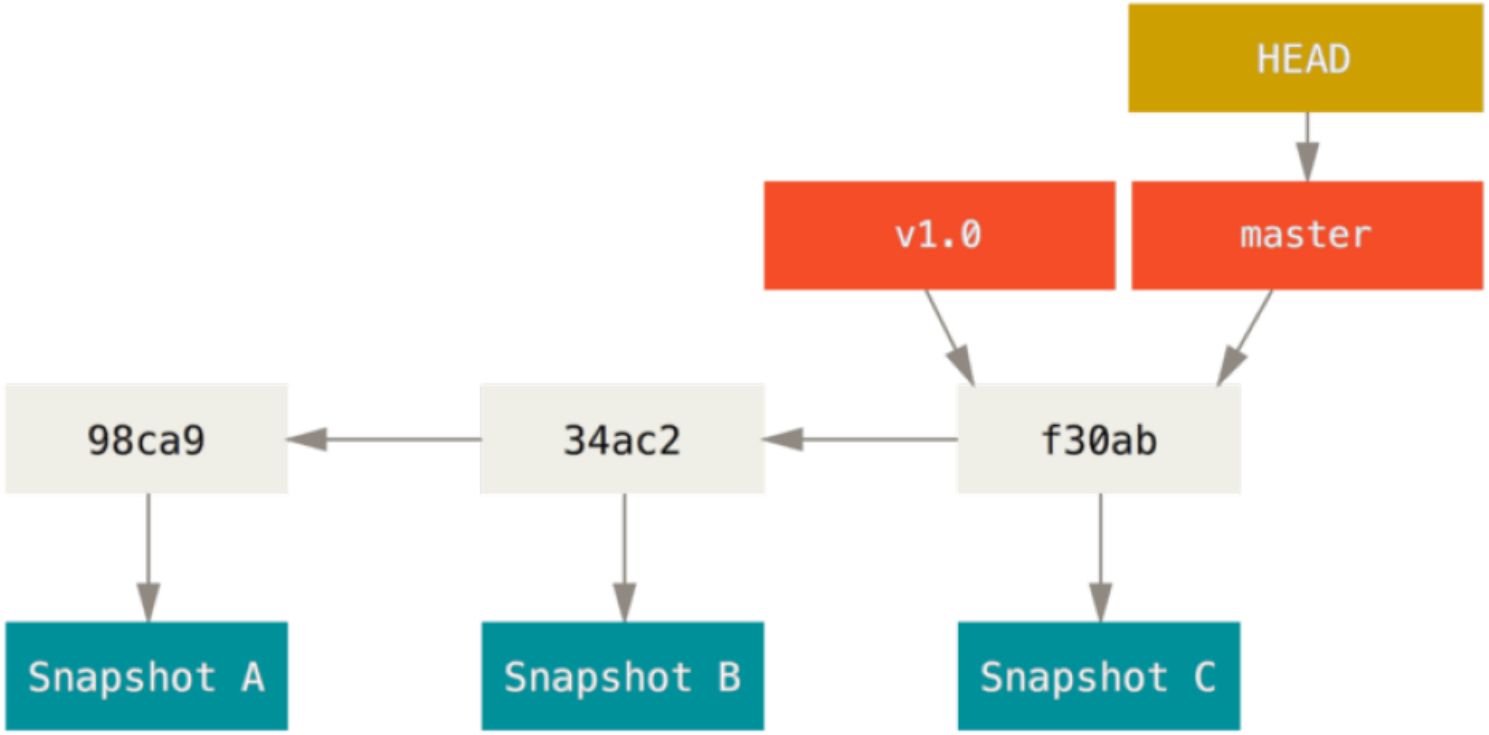
> git add README test.rb LICENSE
> git commit –m "The initial commit of my project

Commit Objects Point Back to Their Parents



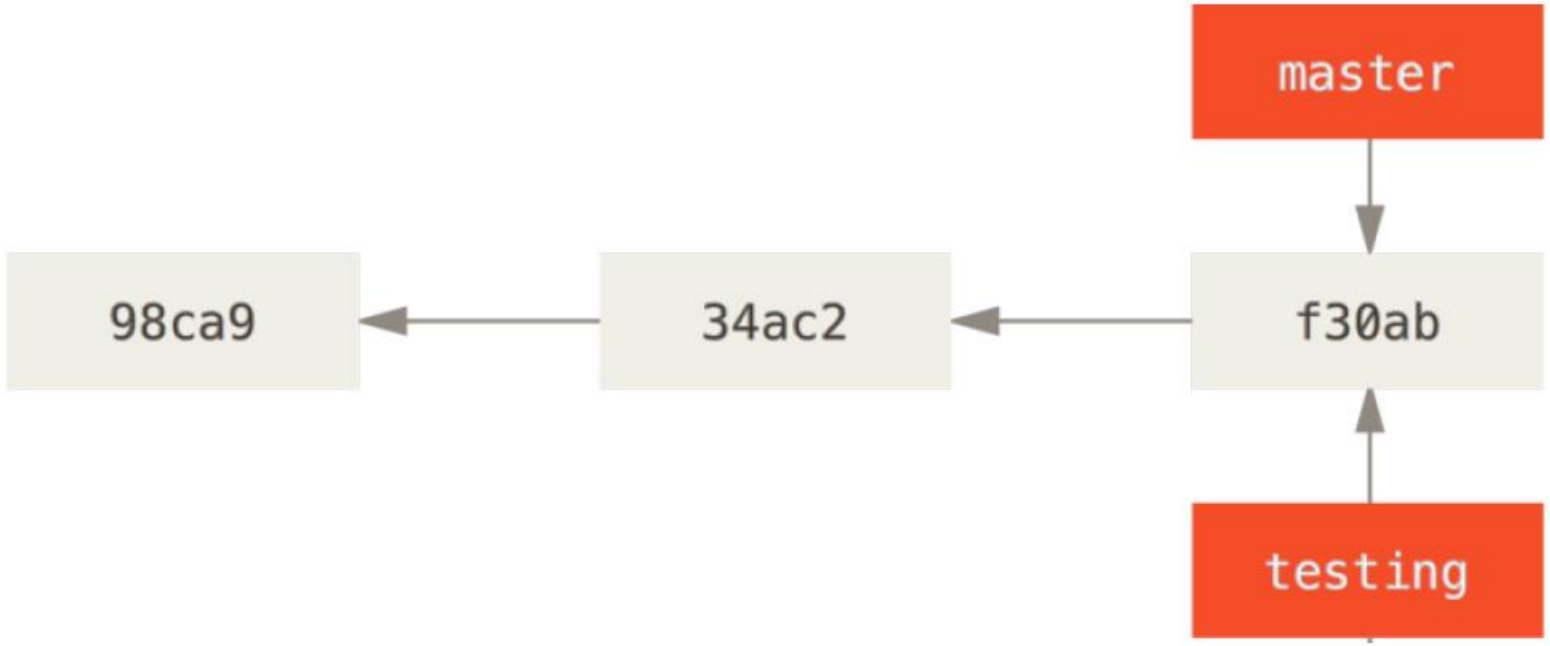
- Next commit stores a pointer to the commit(s) that came before it

A Branch is a Pointer to a Commit Object



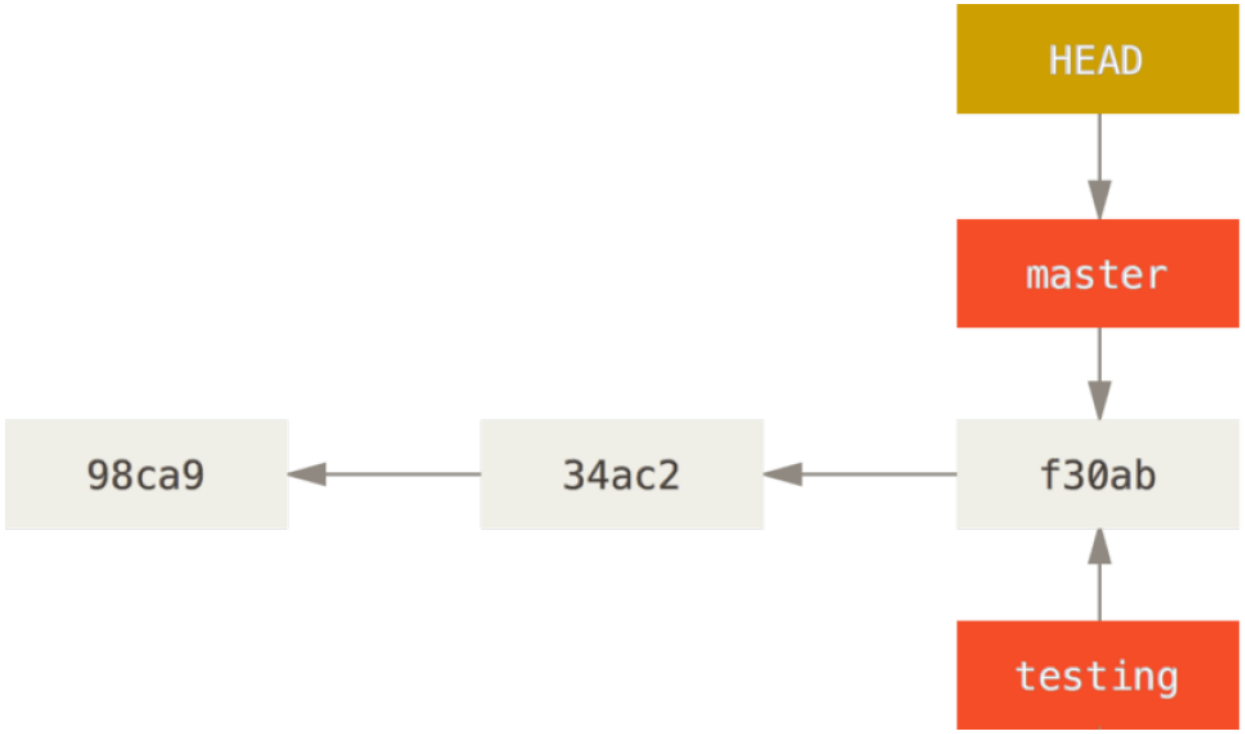
- A *branch* in Git is a lightweight movable pointer to one of these commits
- The default branch in Git is *master*.

Branch Example



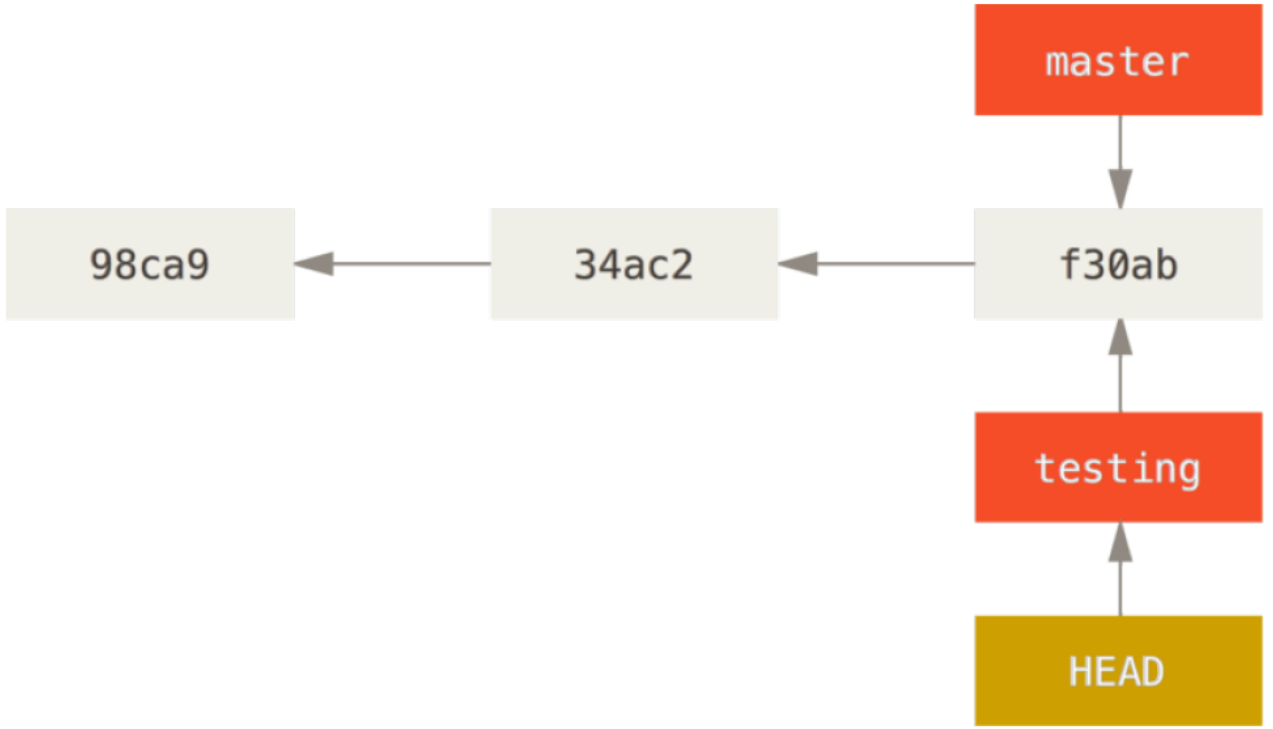
- > git branch testing
- Creates a new pointer to the same commit you're currently on

Branch Example



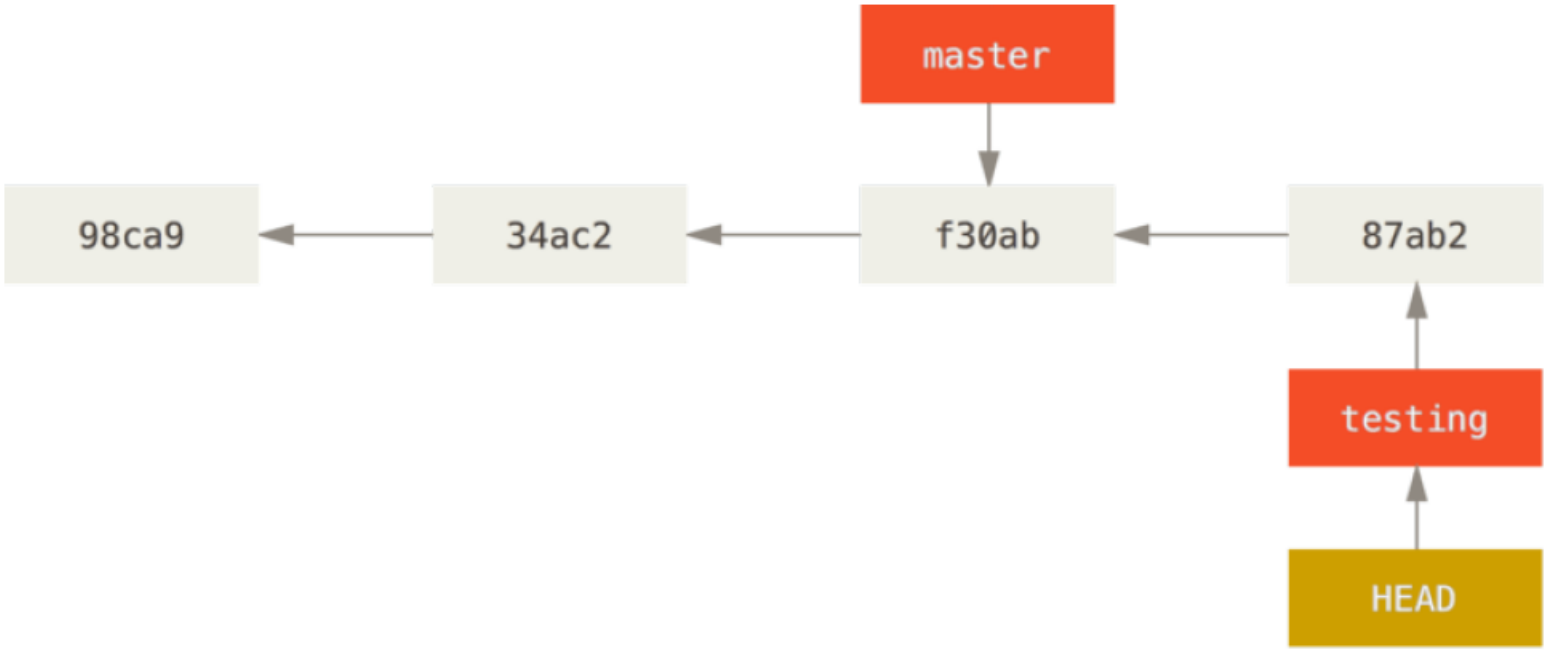
- The HEAD pointer tells you which branch you're currently on
- Currently still on master

Branch Example



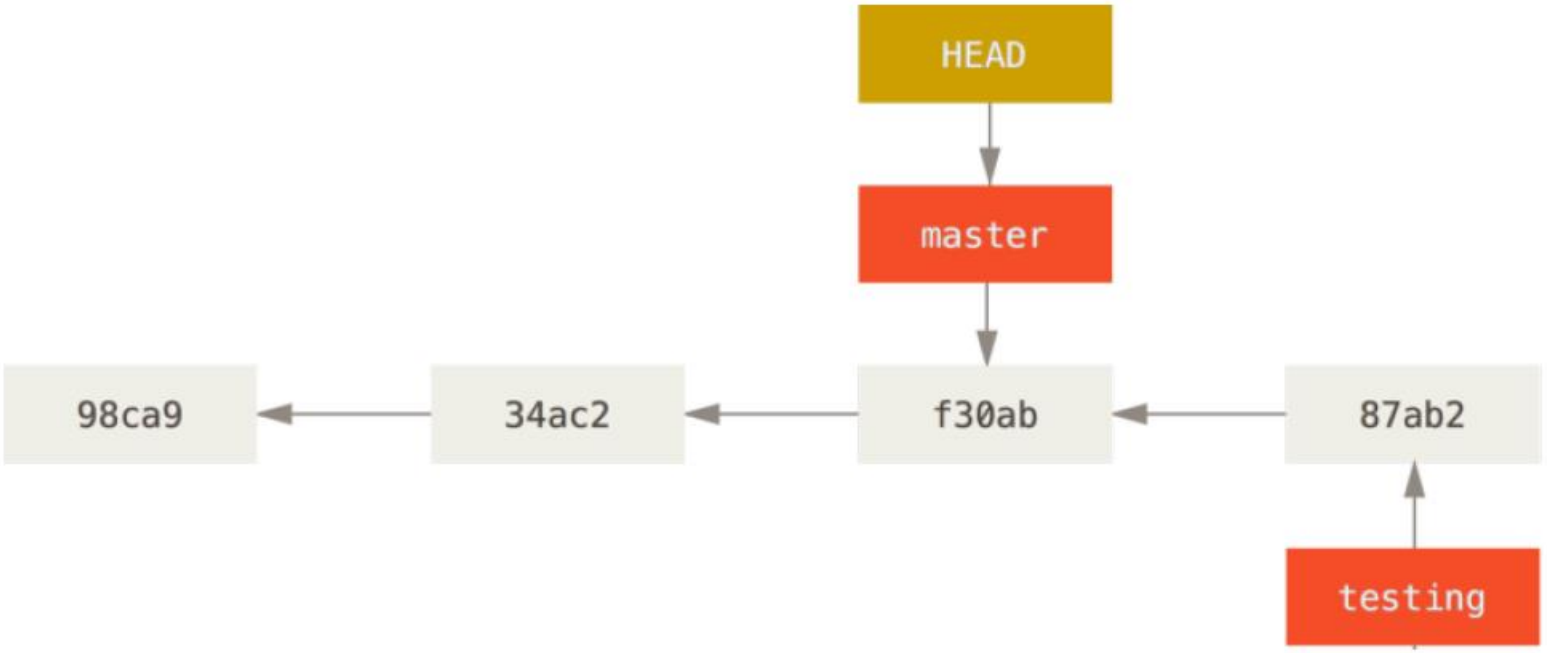
- > git checkout testing
- Switches HEAD pointer to point to an existing branch

Branch Example



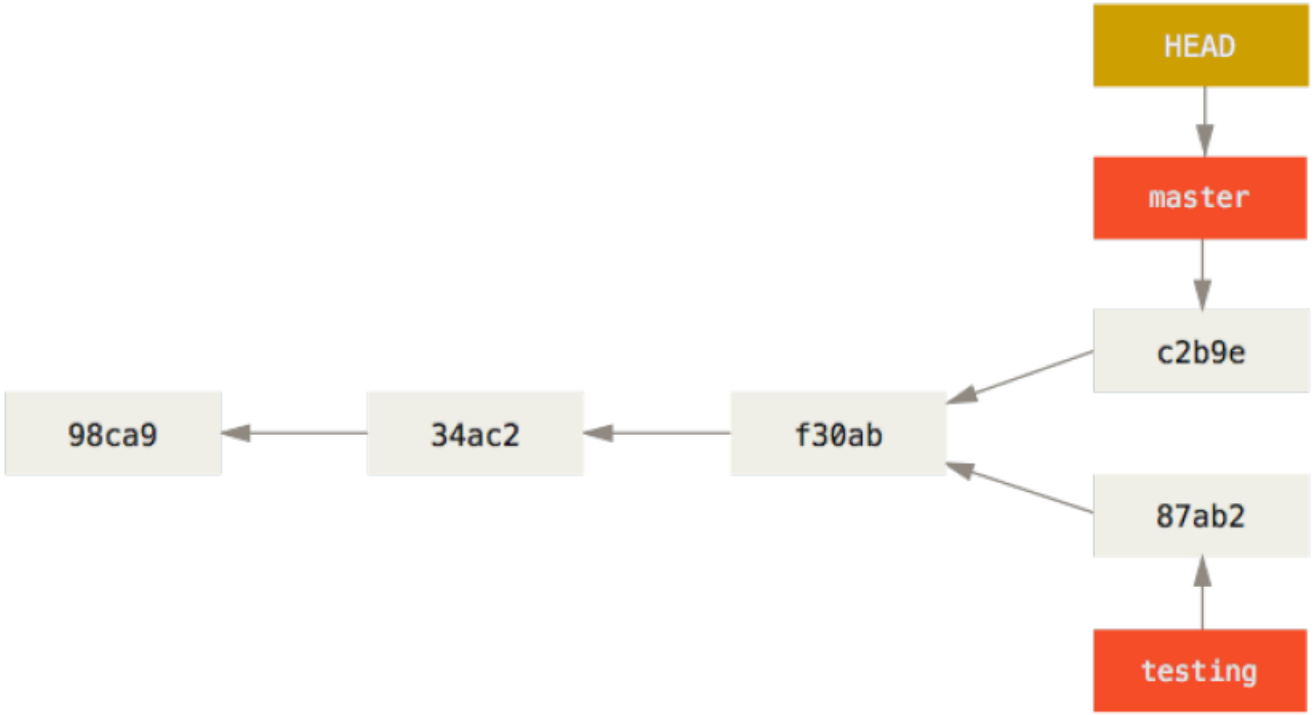
- > git commit –a –m "made some change"
- Next commit moves the testing branch forward

Branch Example



- > git checkout master
- Moves the HEAD pointer back to the master and reverts your files in the working directory to the master branch

Branch Example



- > git commit –a –m "more changes to master"
- Changes now isolated in separate branches

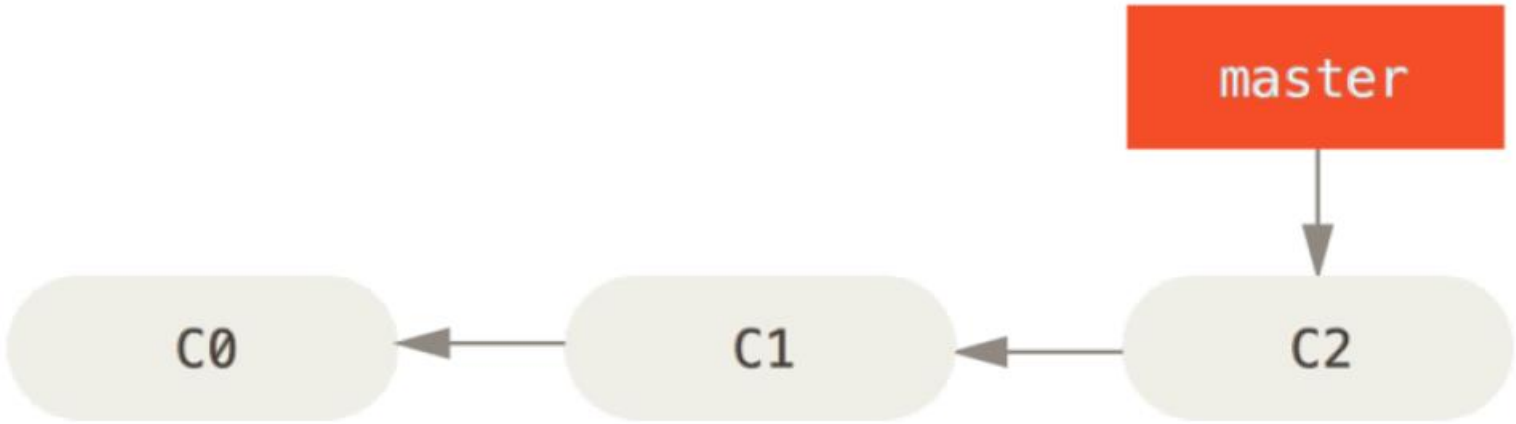
Basic Branching and Merging

- An example workflow
  1. Do work on a website
  2. Create a branch for the new story you're working on
  3. Do some work in the new branch

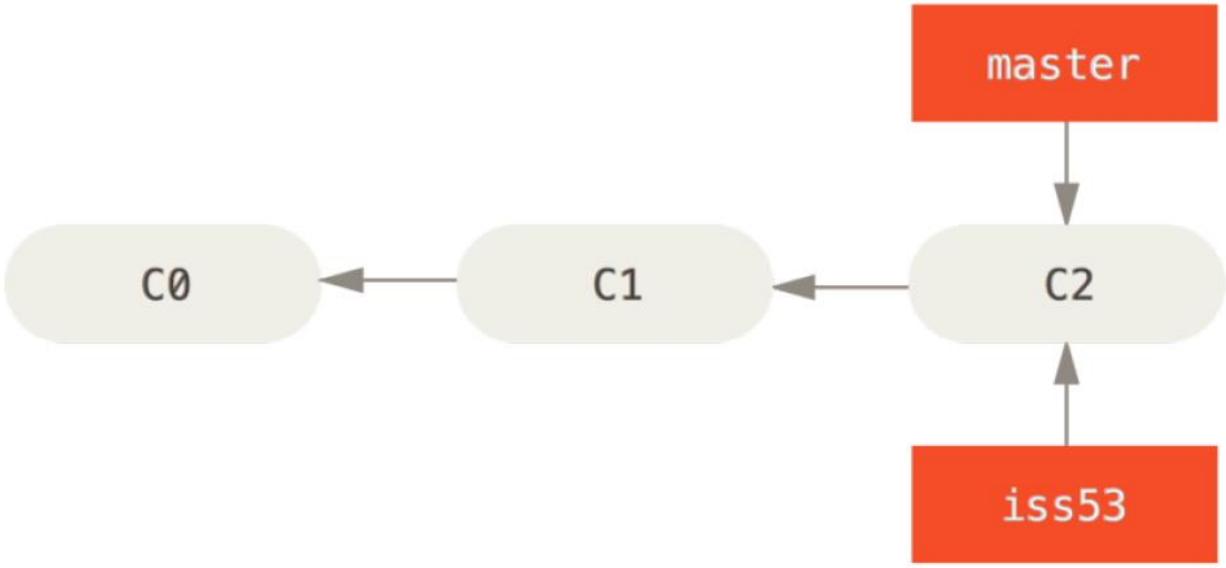
  -- A critical issue needs a hotfix --

  1. Switch to the production branch
  2. Create a branch to add the hotfix
  3. After testing, merge the hotfix branch, and push to production
  4. Switch back to the original story and continue working

Branch and Merge Example

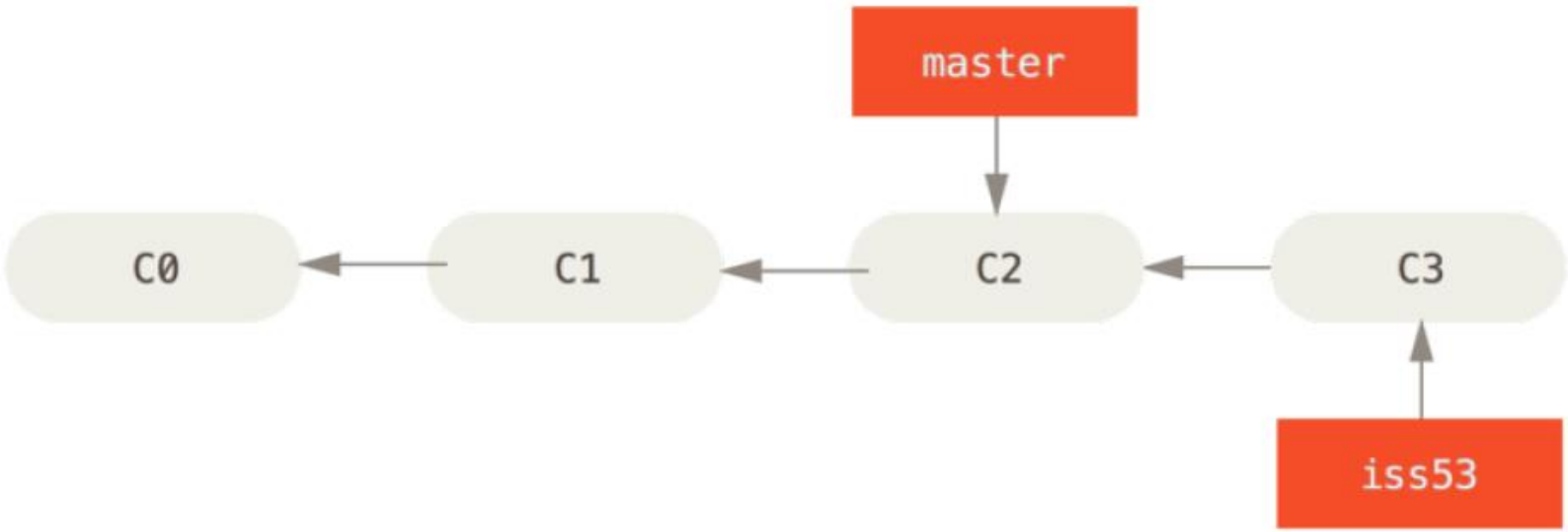


Branch and Merge Example



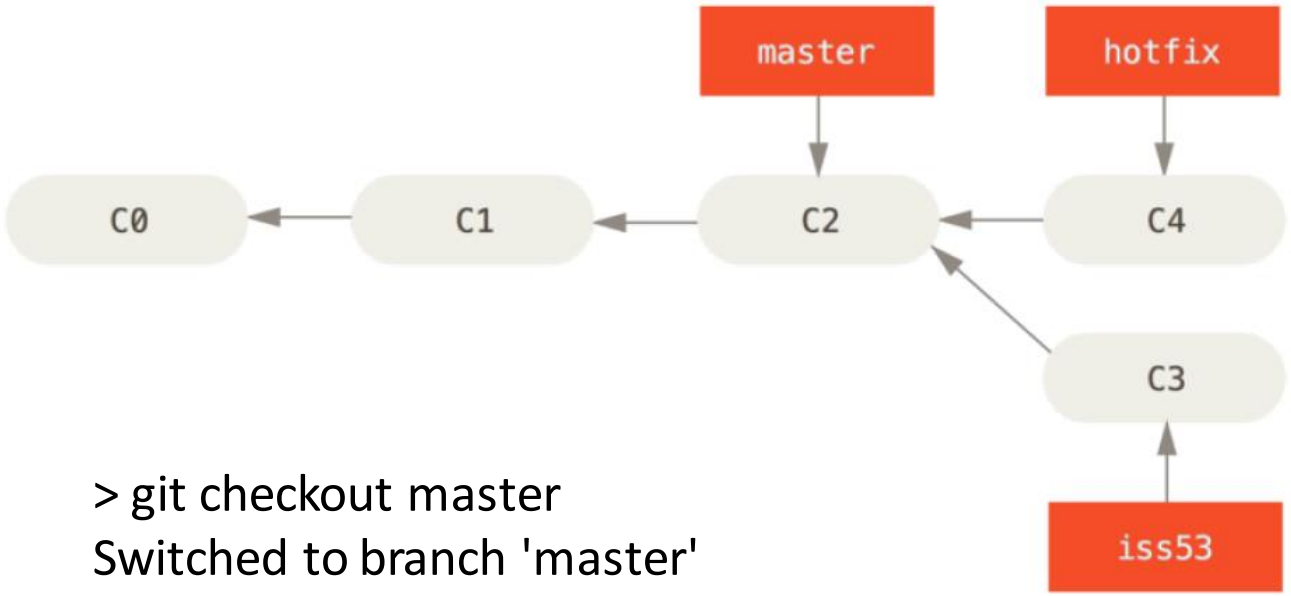
> git checkout –b iss53
Switched to a new branch "iss53"

Branch and Merge Example



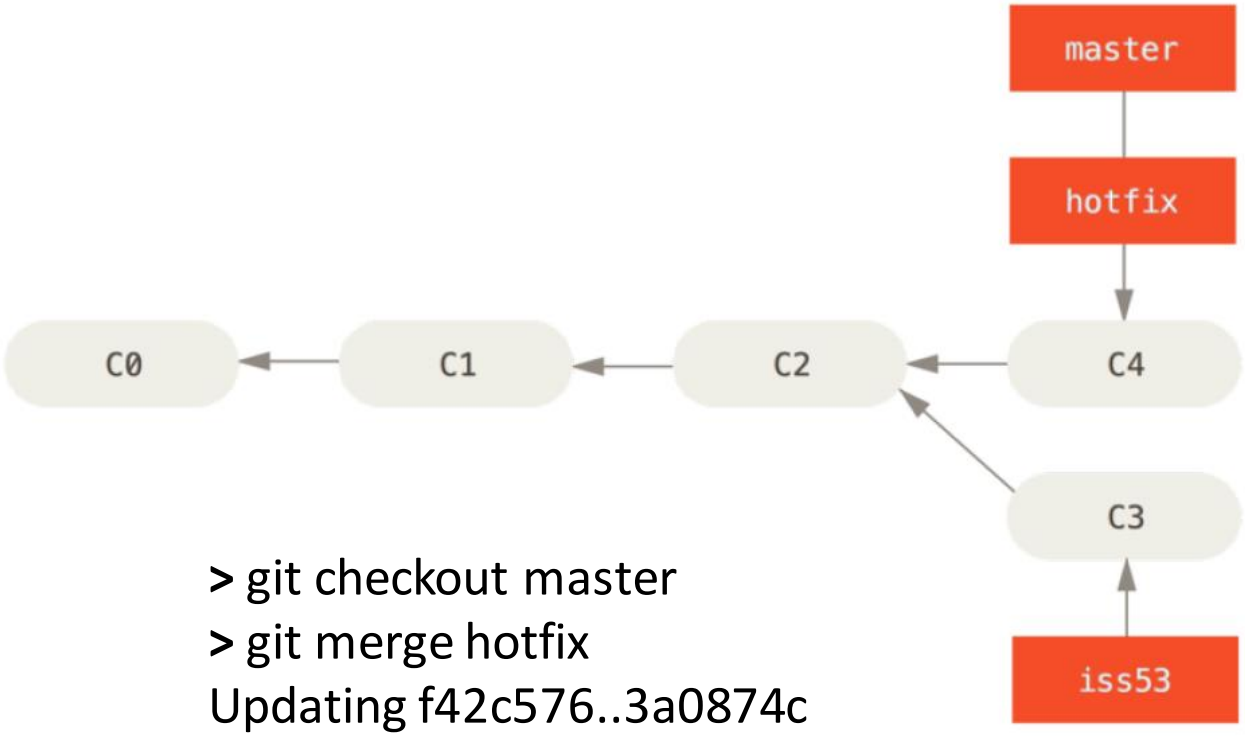
> vim index.html
> git commit -a -m 'added a new footer [issue 53]'

Branch and Merge Example



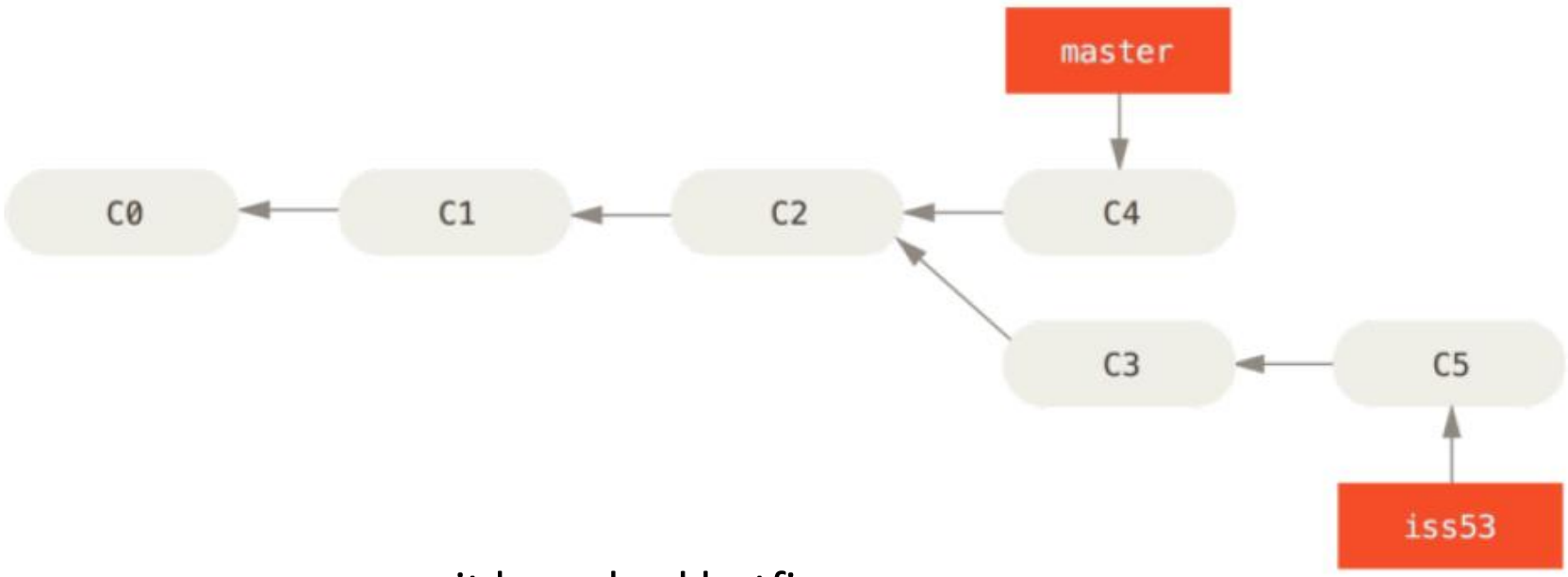
> git checkout master
Switched to branch 'master'
> git checkout -b hotfix
Switched to a new branch 'hotfix'
> vim index.html
> git commit -a -m 'fixed the broken email address'

Branch and Merge Example



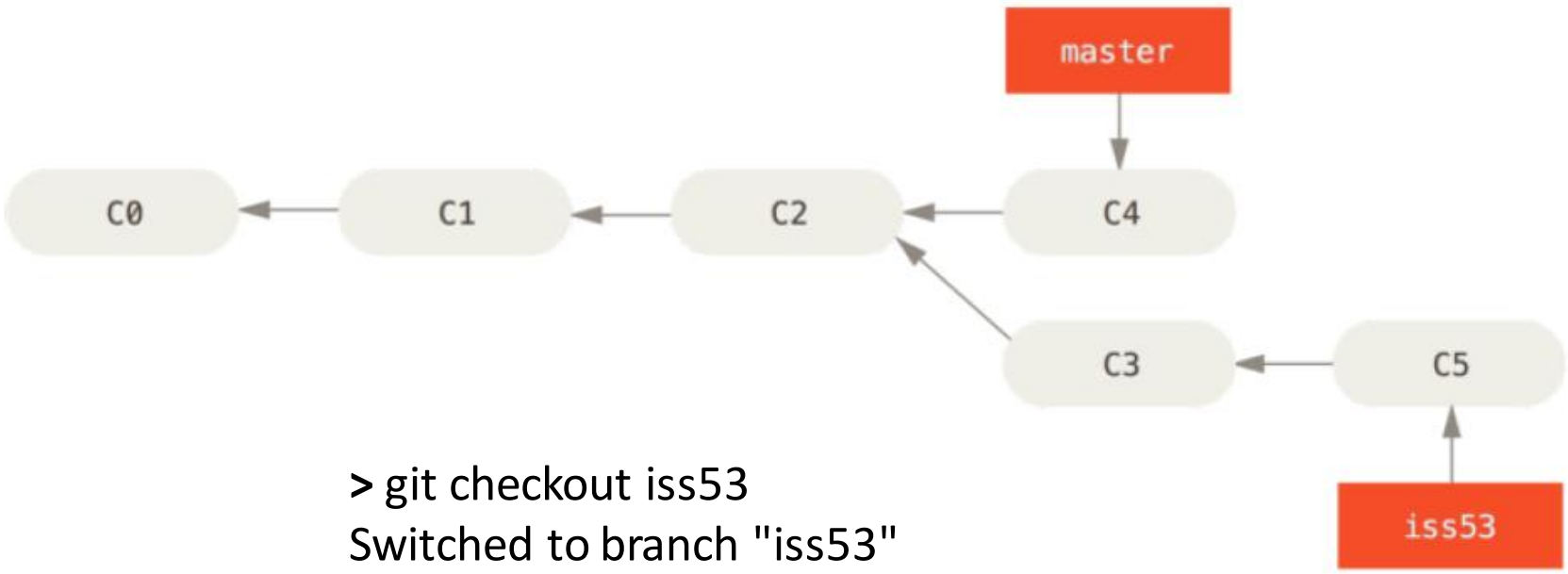

```
> git checkout master
> git merge hotfix
Updating f42c576..3a0874c
Fast-forward
index.html | 2 ++
1 file changed, 2 insertions(+)
```

Branch and Merge Example



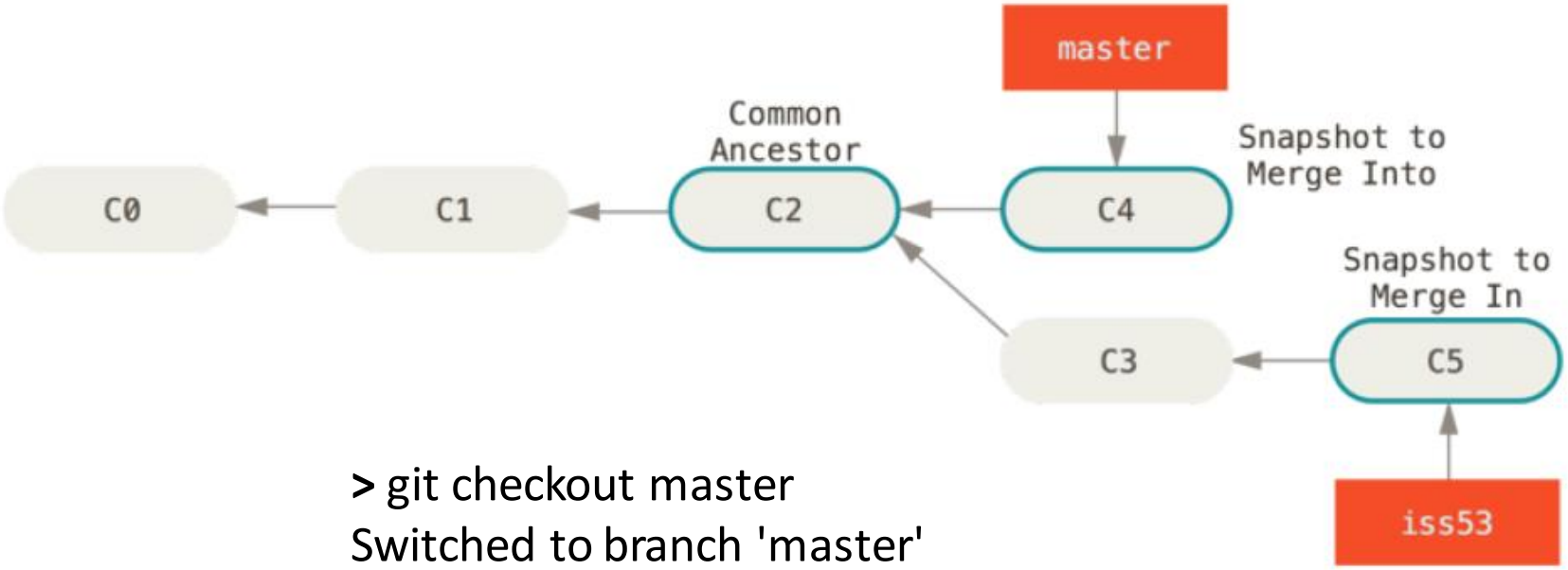
> git branch -d hotfix
Deleted branch hotfix (3a0874c).

Branch and Merge Example



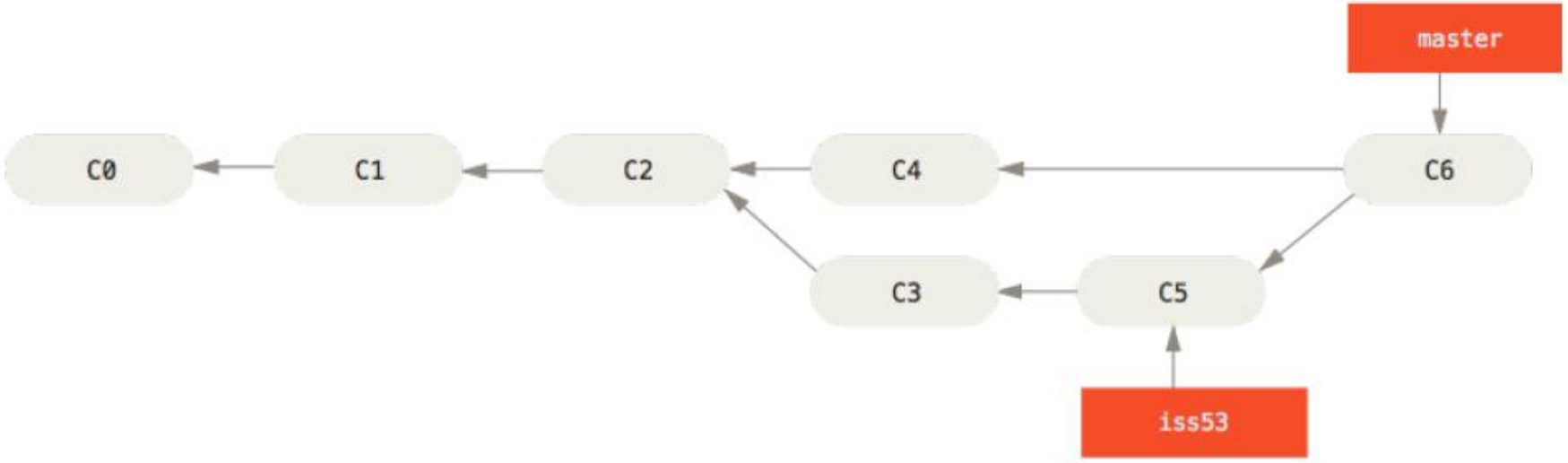

```
> git checkout iss53
Switched to branch "iss53"
> vim index.html
> git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
1 file changed, 1 insertion(+)
```

Branch and Merge Example




```
> git checkout master
Switched to branch 'master'
> git merge iss53
Merge made by the 'recursive' strategy.
index.html | 1 +
1 file changed, 1 insertion(+)
```

Branch and Merge Example



- Result of three-way merge stored in new commit (C6)
- Delete iss53 branch after merge is complete

Merge Conflicts

- If you try to merge two branches with different changes to the same parts of the same file, git will report a merge conflict

```
> git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Merge Conflicts

- When a conflict occurs, git pauses the commit process

```
> git status
On branch master
You have unmerged paths.
(fix conflicts and run "git commit")

Unmerged paths:
(use "git add <file>..." to mark resolution)

        both modified: index.html

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

Merge Conflicts

- Git adds standard conflict resolution markers:

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Merge Conflicts

- Conflicts should be resolved manually
  - There are tools to assist with merging ('git mergetool')
  - In our example, the resolved code might be:

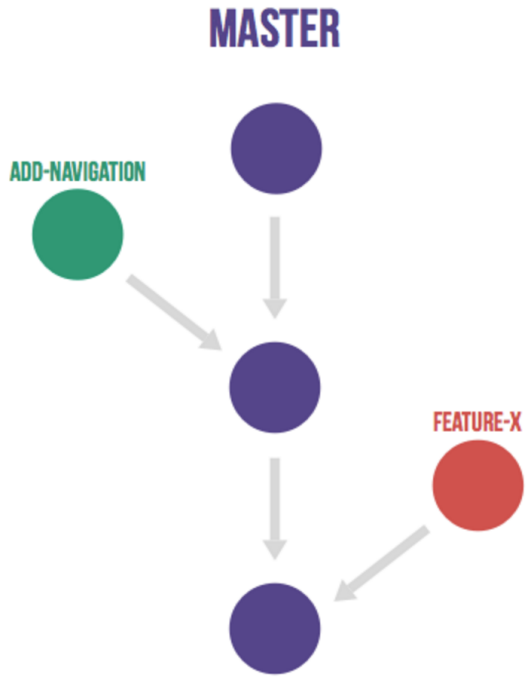
```
<div id="footer">
please contact us at email.support@github.com
</div>
```

- To resolve conflicts, add the conflicted file(s) to the staging area
  - > git add index.html
- To complete the merge, commit the resolved files
  - > git commit –m "merge commit"

Git Workflow

- A standard set of best practices for developing a project with git
  - Includes development model for branching / merging / deploying code
  - Encourages good development practices (feature-driven development, code reviews, continuous delivery)
  - Can be applied to large development teams or across different projects

GitHub Flow



- Anything in the `master` branch is deployable
- To work on something new, create a descriptively named branch off of `master`
- Commit to that branch locally and regularly push your work to the same named branch on the server
- When you need feedback or help, or you think the branch is ready for merging, open a merge / pull request
- After someone else has reviewed and signed off on the feature, you can merge it into `master`
- Once it is merged and pushed to `master`, you can deploy immediately