

Intro to Git

Plan

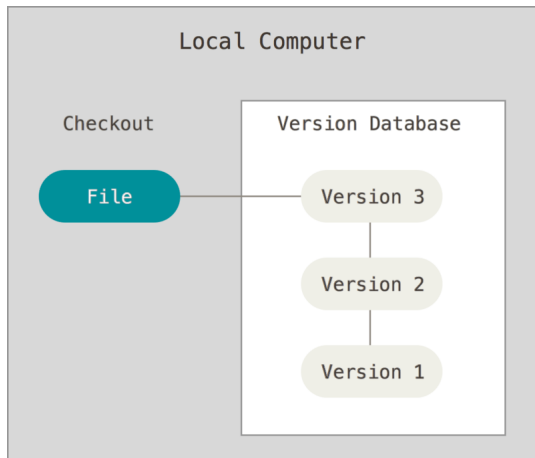
- ▶ Quick Intro to Version Control
- ▶ Getting Started
- ▶ Git Basics
 - ▶ Recording Changes
 - ▶ Staging
 - ▶ Committing
 - ▶ Viewing History
- ▶ Branches in Git
 - ▶ Branches
 - ▶ Merging
 - ▶ Rebasing
- ▶ Slides based off <https://git-scm.com/book/en/v2>

Quick Intro to Version Control

- ▶ VCS: Version Control System
- ▶ System that records changes to a file or set of files over time
- ▶ Can recall specific versions at a later time
 - ▶ Revert individual files or even the entire project to a previous state
 - ▶ Compare changes in files over time
 - ▶ See who modified something and when
 - ▶ Do the above efficiently
- ▶ Set of all versions of all files called a repository

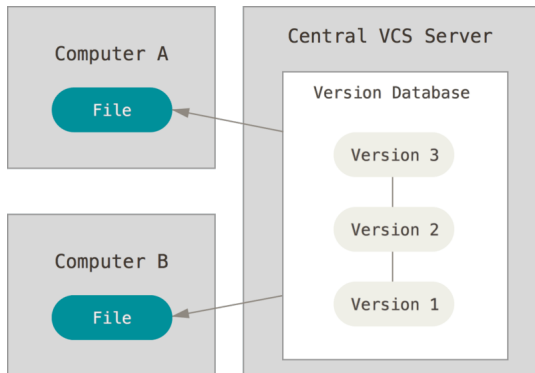
Local VCS

- ▶ Simple database containing all changes to file under version control
- ▶ You "check out" versions of the project history



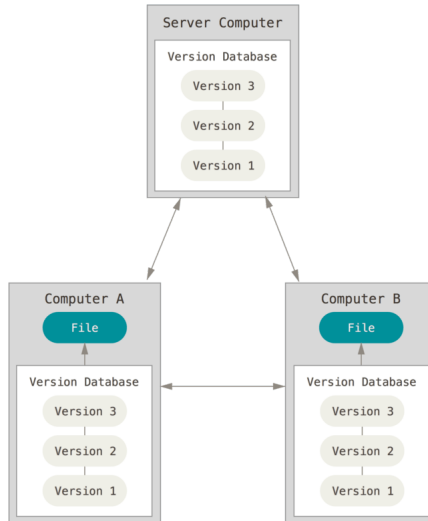
Centralised VCS

- ▶ Used to collaborate with other developers
- ▶ Single server contains all the versioned files



Distributed VCS

- ▶ Each client fully mirrors the the repository
- ▶ Allows direct collaboration between developers

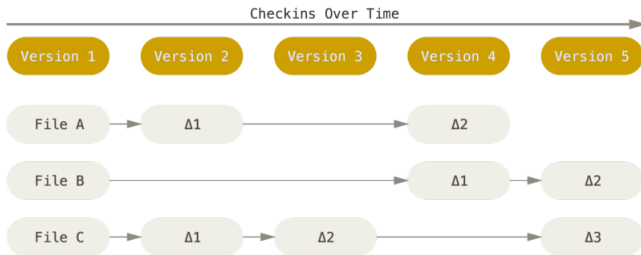


A Short History of Git

- ▶ Created by Linus Torvalds in 2005 for Linux kernel development
- ▶ The goals for Git were:
 - ▶ Speed
 - ▶ Simple design
 - ▶ Able to handle large projects
 - ▶ Fully distributed
 - ▶ Very good support for non-linear development (branches)

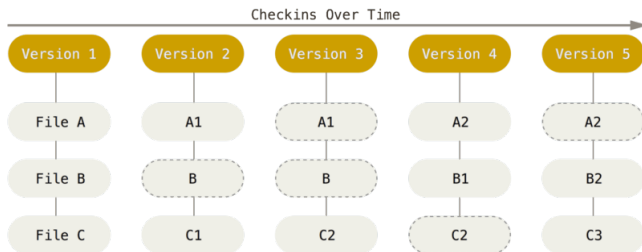
Differences

- ▶ Store initial file version and each change over time
- ▶ Subversion uses this method



Snapshots

- ▶ This is the method Git uses
- ▶ Every time you commit (save) the project state, a new snapshot of is made
- ▶ A snapshot is a "picture" of all the files in the repo at that time
- ▶ Files that haven't changed aren't saved for efficiency



Almost everything in Git is local

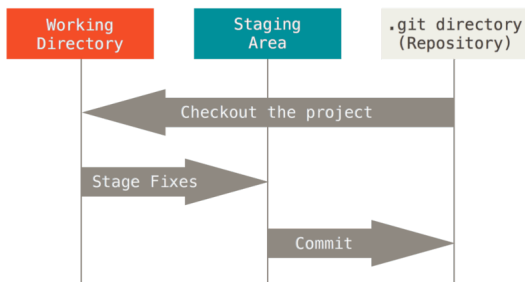
- ▶ Most operations in Git only affect your local copy of the repo
- ▶ Very rarely need to go onto the network
- ▶ No network latency
- ▶ Can work offline

Git has built-in integrity checking

- ▶ Everything stored in Git is check-summed
- ▶ Changing something changes the checksum and the history
- ▶ Git is able to detect file corruption or modification
- ▶ SHA-1 hash (40-char hex string)
- ▶ 24b9da6552252987aa493b52f8696cd6d3b00373

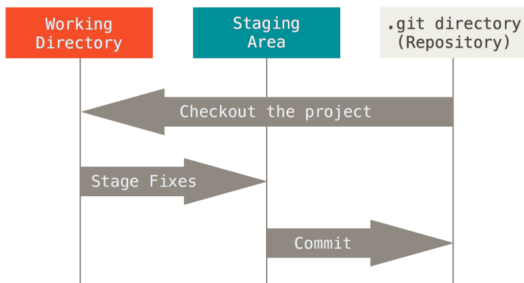
The Three States

- ▶ One of the most important things in Git
- ▶ Files in your repository exist in one of three states
- ▶ Committed: Safely stored in your local database
- ▶ Modified: Changed a file but not committed it yet
- ▶ Staged: Marked a modified file to go in the next commit snapshot



The Three States

- ▶ The .git directory is where Git stores the snapshots
- ▶ The working tree is a single snapshot of the repository
- ▶ Uncompressing a snapshot from .git is called "checking out"
- ▶ The staging area stores information about what goes into the next commit



Development Cycle

- ▶ 1. Modify files in working tree
- ▶ 2. Stage some of the modified files
- ▶ 3. Do a commit, saving the contents of the staging area into a new snapshot

Installing Git

- ▶ Available on Linux, Windows, Mac
- ▶ Package managers or from source at <https://github.com/git/git/releases>

First Time Configuration

- ▶ Configuration in git is done with the **git config** command
- ▶ This command allows you to get and set configuration variables
- ▶ These variables can be stored in three different places
- ▶ */etc/gitconfig* file: System-wide values, use `--system` switch
- ▶ *~/.gitconfig* or *~/.config/git/config* file: This user only, use `--global` switch
- ▶ *.git/config* file: This repository only, no switch required (default option)

First Time Configuration

- ▶ **git config --global user.name "Sam Caulfield"**
- ▶ **git config user.email "sam.caulfield@movidius.com"**
- ▶ **git config --system core.editor "vim"**
- ▶ Precedence: Per Repository > Per User > System
- ▶ Show configuration settings with **git config --list**
- ▶ Show just one configuration option with **git config user.name**

Getting help

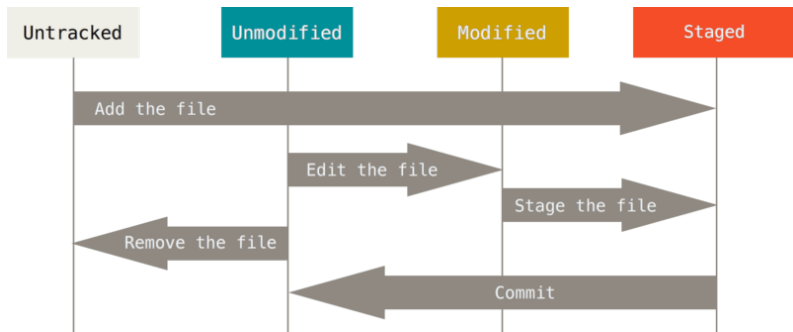
- ▶ **git help <verb>**
- ▶ **git <verb> --help**
- ▶ **man git-<verb>**
- ▶ <https://git-scm.com/book/en/v2/>

Getting a Git Repository

- ▶ Two main ways: create one or copy one
- ▶ To create one: **git init** or **git init MyRepo**
- ▶ To copy one: **git clone <https://github.com/libgit2/libgit2>**

Recording Changes to the Repository

- ▶ Need to make changes and commit snapshots
- ▶ Files in the working directory are either tracked or untracked
- ▶ Tracked: Present in the previous snapshot
- ▶ Untracked: Not present in the previous snapshot or in the staging area
- ▶ Tracked files can be unmodified, modified, or staged



Checking the Status of Files

- Done with the **git status** command

```
sc@u: ~/MyRepo
sc@u:~/MyRepo$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
sc@u:~/MyRepo$
```

```
sc@u: ~/MyRepo
sc@u:~/MyRepo$ touch README
sc@u:~/MyRepo$ git status
On branch master

Initial commit

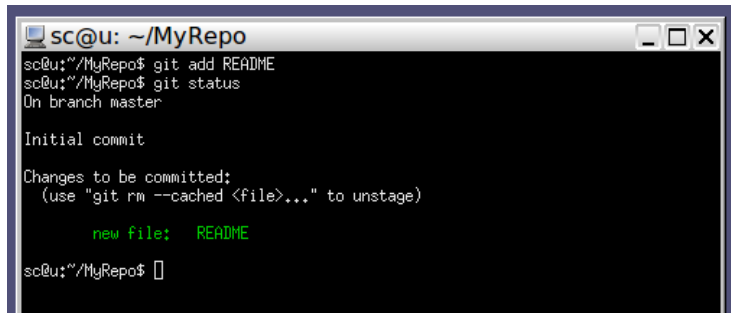
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        README

nothing added to commit but untracked files present (use "git add" to track)
sc@u:~/MyRepo$
```

Tracking New Files

- ▶ Done with the **git add** command
- ▶ This *stages* the file, which makes it become tracked

A terminal window titled 'sc@u: ~/MyRepo' with standard window controls. The terminal shows the execution of 'git add README' followed by 'git status'. The status output indicates an initial commit on the master branch with changes to be committed, specifically a new file named README.

```
sc@u: ~/MyRepo$ git add README
sc@u: ~/MyRepo$ git status
On branch master

Initial commit

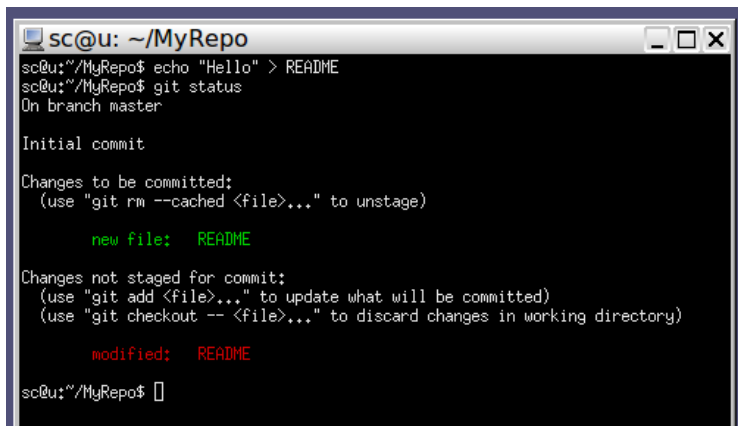
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README

sc@u: ~/MyRepo$
```

Staging Modified Files

- ▶ We modified the file
- ▶ The new changes aren't automatically staged
- ▶ Now the file has staged modifications and unstaged modifications

A terminal window with a title bar that says 'sc@u: ~/MyRepo'. The window contains the following text:

```
sc@u:~/MyRepo$ echo "Hello" > README
sc@u:~/MyRepo$ git status
On branch master

Initial commit

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

    new file:   README

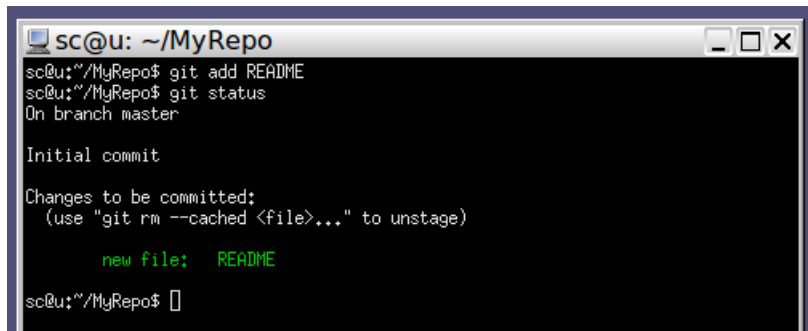
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README

sc@u:~/MyRepo$
```

Staging Modified Files

- ▶ Done with the **git add** command
- ▶ **git add --patch** can be used to stage parts of files

A terminal window with a title bar that reads "sc@u: ~/MyRepo". The window contains the following text:

```
sc@u:~/MyRepo$ git add README
sc@u:~/MyRepo$ git status
On branch master

Initial commit

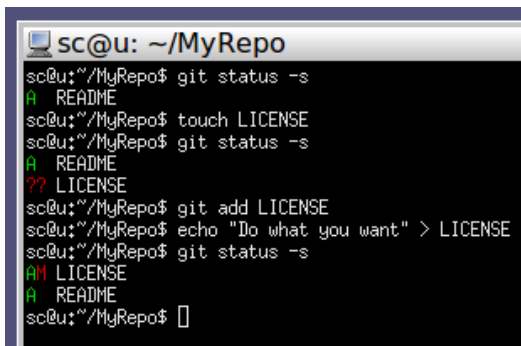
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   README

sc@u:~/MyRepo$
```


Short Status

- ▶ **git status -s** provides a less verbose status report
- ▶ Untracked files have a `??` next to them
- ▶ New files that have been added to the staging area have an `A`
- ▶ Modified files have an `M`, etc.
- ▶ Left column: index, right column: working directory

A terminal window titled 'sc@u: ~/MyRepo' showing the execution of 'git status -s' at three different points. The first run shows 'A README'. The second run, after creating 'LICENSE', shows 'A README' and '?? LICENSE'. The third run, after adding 'LICENSE' to the staging area, shows 'AM LICENSE' and 'A README'.

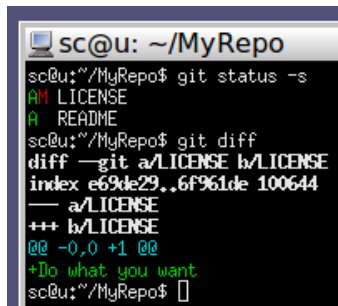
```
sc@u:~/MyRepo$ git status -s
A  README
sc@u:~/MyRepo$ touch LICENSE
sc@u:~/MyRepo$ git status -s
A  README
?? LICENSE
sc@u:~/MyRepo$ git add LICENSE
sc@u:~/MyRepo$ echo "Do what you want" > LICENSE
sc@u:~/MyRepo$ git status -s
AM LICENSE
A  README
sc@u:~/MyRepo$
```

Ignoring Files

- ▶ The working directory can easily become full of files you don't want to version
- ▶ .o, .swp, .log, etc.
- ▶ They can clog up the output of **git status**
- ▶ List unwanted file types in a .gitignore file in the repository
- ▶ **echo *.o > .gitignore**
- ▶ Can perform simple pattern matching: doc/**/*.pdf

Viewing Staged and Unstaged Changes

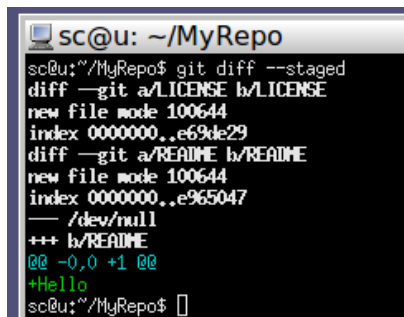
- ▶ What have I changed but not staged?
- ▶ What have I staged that I am about to commit?
- ▶ Use the **git diff** command
- ▶ This shows what's changed in the working directory that isn't staged
- ▶ i.e. it shows the *difference* between the staging area and working directory



```
sc@u: ~/MyRepo
sc@u:~/MyRepo$ git status -s
AM LICENSE
A  README
sc@u:~/MyRepo$ git diff
diff --git a/LICENSE b/LICENSE
index e69de29..6f961de 100644
--- a/LICENSE
+++ b/LICENSE
@@ -0,0 +1 @@
+Do what you want
sc@u:~/MyRepo$
```

Viewing Staged and Unstaged Changes

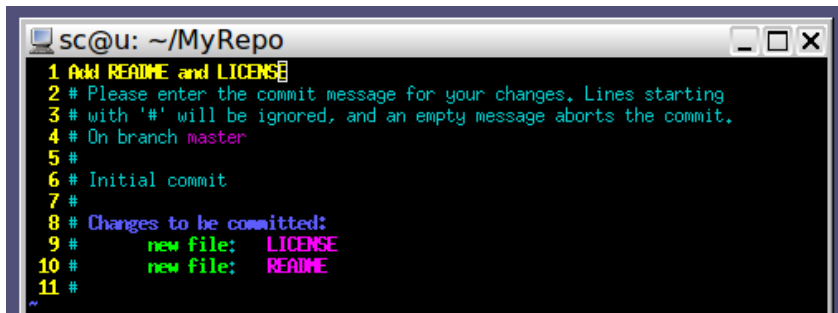
- ▶ To view the exact changes in the staging area use **git diff --staged**

A terminal window with a dark background and light text. The title bar shows a computer icon and the text 'sc@u: ~/MyRepo'. The terminal content shows the command 'git diff --staged' and its output, which includes file mode information, index hashes, and a diff for a file named README, showing a new line '+Hello'.

```
sc@u: ~/MyRepo
sc@u:~/MyRepo$ git diff --staged
diff --git a/LICENSE b/LICENSE
new file mode 100644
index 0000000..e69de29
diff --git a/README b/README
new file mode 100644
index 0000000..e965047
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+Hello
sc@u:~/MyRepo$
```

Committing Your Changes

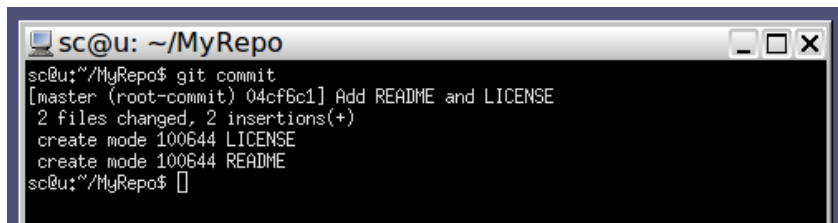
- ▶ Committing creates a new snapshot in the project history
- ▶ The snapshot is the previous snapshot + the staging area changes
- ▶ Anything left in the working directory and not in the staging area isn't included
- ▶ Use the **git commit** command

A terminal window titled 'sc@u: ~/MyRepo' with standard window controls. The terminal displays the output of a 'git commit' command. It shows a numbered list of instructions for entering a commit message, followed by a summary of changes to be committed: 'new file: LICENSE' and 'new file: README'. The cursor is on line 11.

```
sc@u: ~/MyRepo
1 Add README and LICENSE
2 # Please enter the commit message for your changes. Lines starting
3 # with '#' will be ignored, and an empty message aborts the commit.
4 # On branch master
5 #
6 # Initial commit
7 #
8 # Changes to be committed:
9 #   new file:   LICENSE
10 #   new file:  README
11 #
```

Committing Your Changes

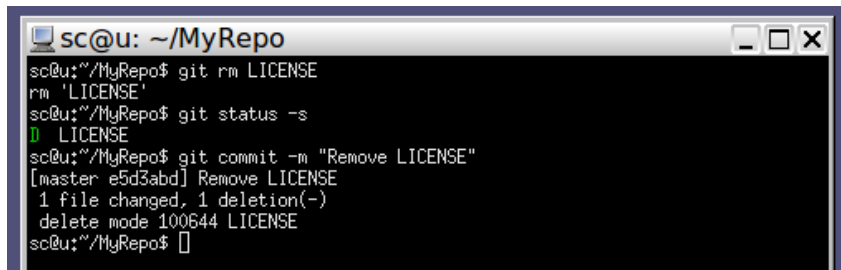
- ▶ Saving and closing the editor confirms the commit
- ▶ Alternatively, you can use **git commit -m "Add README and LICENSE"**
- ▶ Using **git commit -a** will add all tracked files to the commit automatically

A terminal window with a dark background and light text. The title bar shows 'sc@u: ~/MyRepo' and standard window controls. The terminal output shows the execution of 'git commit', the commit message '[master (root-commit) 04cf6c1] Add README and LICENSE', and details about two files being added: LICENSE and README, both with mode 100644.

```
sc@u: ~/MyRepo
sc@u:~/MyRepo$ git commit
[master (root-commit) 04cf6c1] Add README and LICENSE
 2 files changed, 2 insertions(+)
 create mode 100644 LICENSE
 create mode 100644 README
sc@u:~/MyRepo$
```

Removing Files from the Repository

- ▶ Use the **git rm** command
- ▶ The removed file will not be in the next snapshot
- ▶ But all previous versions in history will be untouched

A terminal window titled 'sc@u: ~/MyRepo' with standard window controls. The terminal shows the following commands and output:

```
sc@u:~/MyRepo$ git rm LICENSE
rm 'LICENSE'
sc@u:~/MyRepo$ git status -s
D LICENSE
sc@u:~/MyRepo$ git commit -m "Remove LICENSE"
[master e5d3abd] Remove LICENSE
1 file changed, 1 deletion(-)
delete mode 100644 LICENSE
sc@u:~/MyRepo$
```

- ▶ Similarly, **git mv** can be used to move files

Viewing the Commit History

- Use the **git log** command

```
sc@sc-ubuntu:~/MyRepo$ git log
commit 3b0ea457b06e0c5f7cd5b814916d8d2e93564d0a
Author: Sam Caulfield <sam.caulfield@movidius.com>
Date: Thu Feb 23 15:32:29 2017 +0000

    Add LICENCE

commit 3ddc097644268072a645935b632106c309d8550c
Author: Sam Caulfield <sam.caulfield@movidius.com>
Date: Thu Feb 23 15:32:47 2017 +0000

    Add CONTRIBUTING

commit e1f1735dce9cd1facca6371fa10c37afd0b694b7
Author: Sam Caulfield <sam.caulfield@movidius.com>
Date: Thu Feb 23 15:32:08 2017 +0000

    Add README
sc@sc-ubuntu:~/MyRepo$
```


Viewing the Commit History

- ▶ Can control the output of **git log**
- ▶ -p: Show diffs in each commit
- ▶ -1: Limit output to last 1 commit

```
sc@sc-ubuntu:~/MyRepo$ git log -p -1
commit 3b0ea457b06e0c5f7cd5b814916d8d2e93564d0a
Author: Sam Caulfield <sam.caulfield@movidius.com>
Date: Thu Feb 23 15:32:29 2017 +0000

    Add LICENCE

diff --git a/LICENCE b/LICENCE
new file mode 100644
index 0000000..e69de29
sc@sc-ubuntu:~/MyRepo$
```

Viewing the Commit History

- Keep logs to one line each: **git log --pretty=oneline**

```
sc@sc-ubuntu:~/MyRepo$ git log --oneline
3b0ea45 Add LICENCE
3ddc097 Add CONTRIBUTING
e1f1735 Add README
sc@sc-ubuntu:~/MyRepo$
```

Viewing the Commit History

- ▶ Restrict output based on time: **git log --since=1.day**
- ▶ See what commits modified a string: **git log -Ssomestring**
- ▶ Filter by author: **git log --author**
- ▶ Filter by commit message content: **git log --grep**

Undoing Things

- ▶ You can amend a commit if you forgot something
- ▶ Use **git commit --amend**
- ▶ Takes your staging area and adds it to the most recent commit
- ▶ Results in a single commit: original + changes
- ▶ Allows you to redo the commit message

Unstaging Staged Files

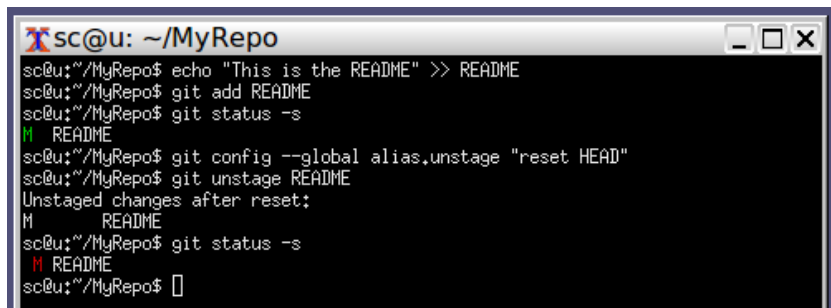
- ▶ Use the **git reset HEAD <file>** command
- ▶ This removes the file from the staging area
- ▶ This is safe because the changes are also in the working directory
- ▶ Warning: **git reset --hard** isn't necessarily safe

Unmodifying Modified Files

- ▶ In Git, "unmodifying" means resetting a file back to the previous snapshot
- ▶ Use the **git checkout -- <file>** command
- ▶ Warning: since uncommitted changes are being removed from the working directory, the changes will be lost unless they are also staged

Aliases

- ▶ Can be used to shorten Git commands
- ▶ Allows you to type "git st" instead of "git status", etc.
- ▶ **git config --global alias.st status**
- ▶ **git config --global alias.unstage 'reset HEAD'**



```
sc@u: ~/MyRepo
sc@u:~/MyRepo$ echo "This is the README" >> README
sc@u:~/MyRepo$ git add README
sc@u:~/MyRepo$ git status -s
M README
sc@u:~/MyRepo$ git config --global alias.unstage "reset HEAD"
sc@u:~/MyRepo$ git unstage README
Unstaged changes after reset:
M README
sc@u:~/MyRepo$ git status -s
M README
sc@u:~/MyRepo$
```

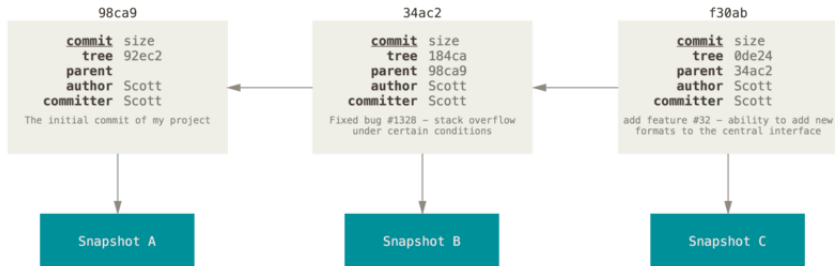
Branches in Git

- ▶ Branching: Diverge from the main line of development
- ▶ Continue working on a different "line" of development
- ▶ Can merge back into the main line when complete
- ▶ One of Git's best features

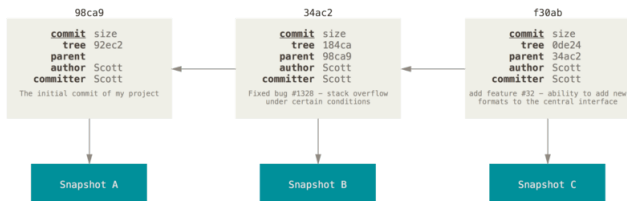
How Git Branches Work

- ▶ When you make a commit, Git stores a commit object
- ▶ The commit object stores a pointer to the snapshot of the staged content
- ▶ Each commit also points to its parent, the one that came before it
- ▶ Only the root commit (the first in the repository) doesn't have a parent
- ▶ Some commits can have multiple parents in the case of merge commits

How Git Branches Work

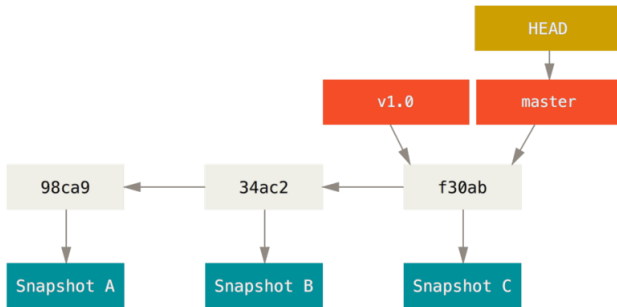


How Git Branches Work



- ▶ A branch in Git is simply a pointer to one of these commits
- ▶ The pointer is moveable
- ▶ In Git, the default branch is called **master**
- ▶ The branch pointer points to the most recent commit on the branch history
- ▶ When you commit on a branch, the branch pointer automatically moves forward

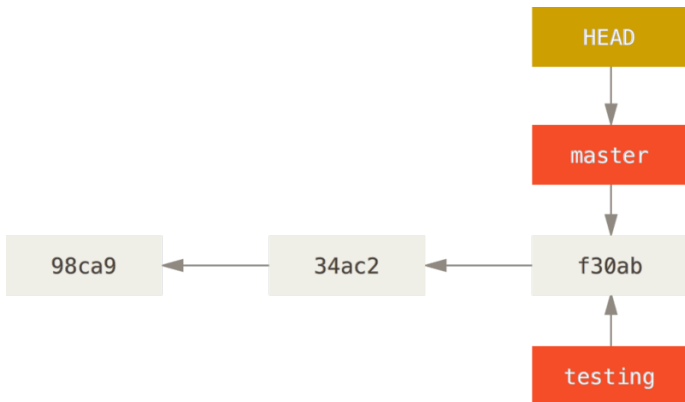
How Git Branches Work



- Branches can be created with **git branch <branchname>**

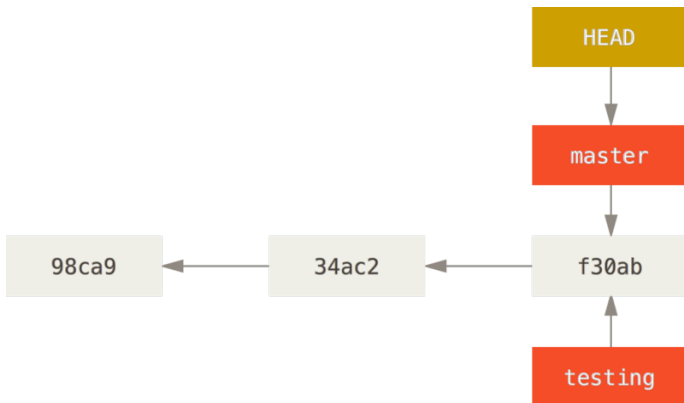
Creating Branches

- ▶ **git branch testing**
- ▶ The new branch points to the commit you are currently on
- ▶ Git uses a special pointer to keep track of what the current branch is
- ▶ This pointer is called HEAD



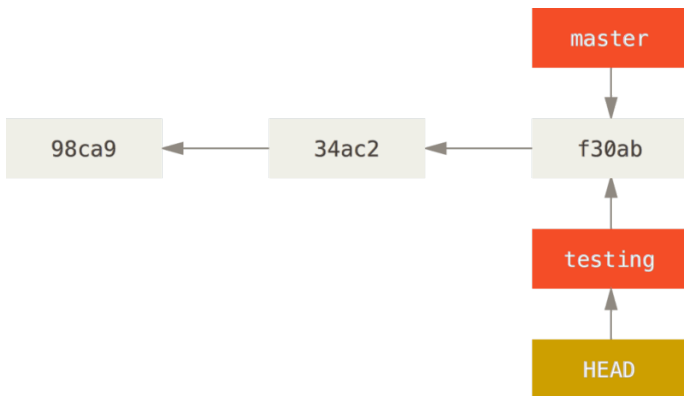
Switching Branches

- ▶ git branch only creates a new branch pointer, it doesn't switch to the branch
- ▶ To switch to another branch, use **git checkout** **<branchname>**
- ▶ **git checkout testing**



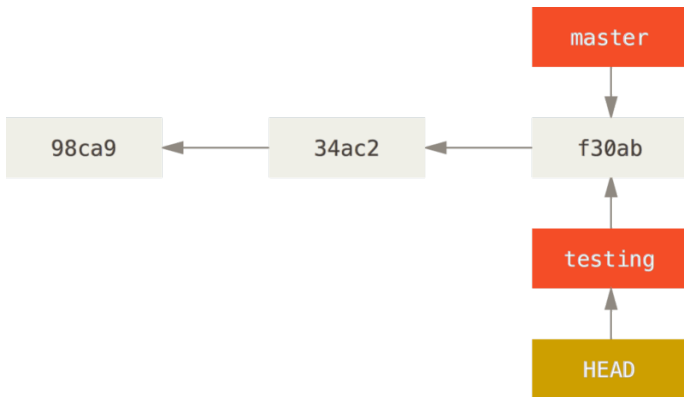
Switching Branches

- ▶ git branch only creates a new branch pointer, it doesn't switch to the branch
- ▶ To switch to another branch, use **git checkout** **<branchname>**
- ▶ **git checkout testing**



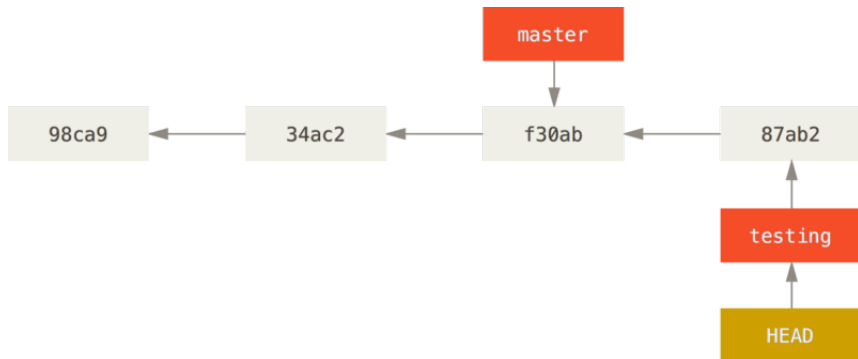
Switching Branches

- ▶ Can also just use **git checkout -b <branchname>**



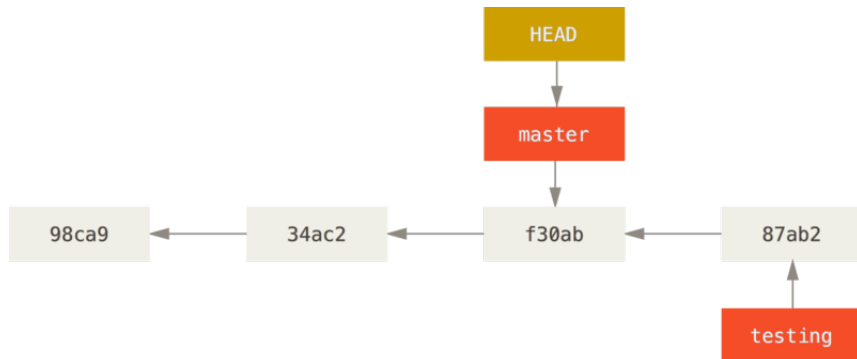
Diverging Branches

- You make a commit on the testing branch..



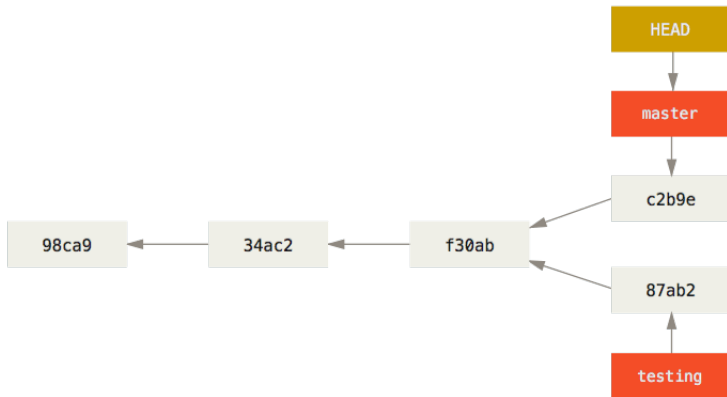
Diverging Branches

- ▶ Then switch back to master with **git checkout master..**



Diverging Branches

- ▶ Then make a commit on master
- ▶ Now the history of master and testing has diverged



Viewing the History with Branches

- ▶ **git log --all --oneline --graph**
- ▶ all: show all branches
- ▶ graph: visually indicate divergent history
- ▶ decorate: show where branch pointers are

```
sc@sc-ubuntu:~/MyRepo$ git log --oneline --graph --all --decorate
* c1535eb (HEAD -> master) Add lib.c
| * b6b24fa (testing) Add Makefile
|/
* df72c15 Add main.c
* f7b4c2a Add LICENSE
* 4120b60 Add README
sc@sc-ubuntu:~/MyRepo$
```

Advantages of Branches in Git

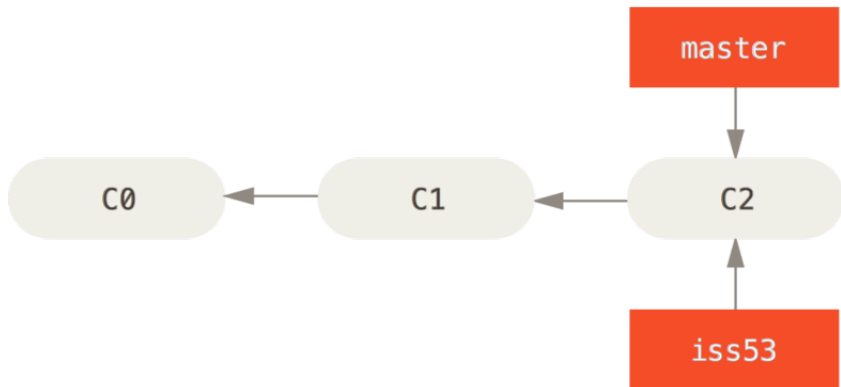
- ▶ In Git, a branch is a file containing a 40-character SHA-1
- ▶ The SHA-1 is the checksum of the commit it points to
- ▶ Creating a new branch in Git = writing 41 bytes to a file

Merging Branches

- ▶ A common use case for branches is "feature branches"
- ▶ You create a new branch for a feature
- ▶ You do the commits for that feature on that branch only
- ▶ Once the feature is done, you merge the feature branch into master

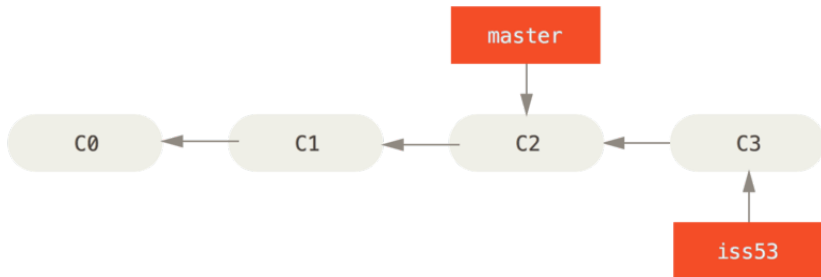
Merging Branches

- Master branch and new branch "iss53" created



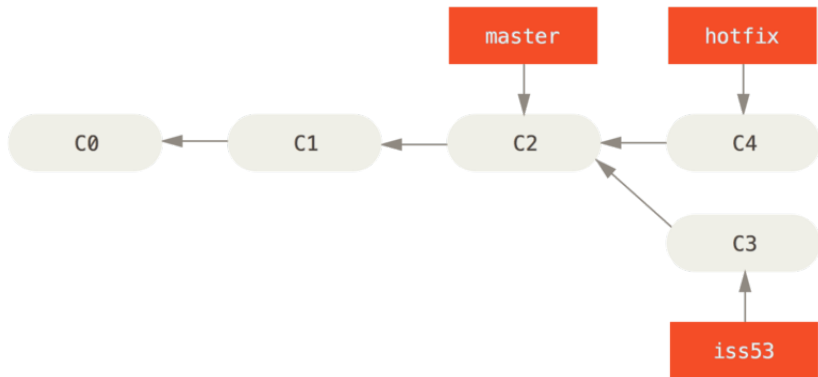
Merging Branches

- ▶ **git checkout iss53**
- ▶ Do some work and **git commit**



Merging Branches

- ▶ Notified of a bug on master branch
- ▶ **git checkout master**
- ▶ **git checkout -b hotfix**
- ▶ Write a fix and **git commit**



Merging Branches

- ▶ Hotfix doesn't exist on master unless we manually merge it
- ▶ Remember commit pointers are unidirectional
- ▶ Master can't see further up the chain
- ▶ To merge in Git, use **git merge <branch>**
- ▶ This merges <branch> into the current branch

```
sc@sc-ubuntu:~/MyRepo$ git checkout master
Switched to branch 'master'
sc@sc-ubuntu:~/MyRepo$ git merge hotfix
Updating a6ef6ac..9c1f2ee
Fast-forward
 README | 1 +
 1 file changed, 1 insertion(+)
sc@sc-ubuntu:~/MyRepo$
```

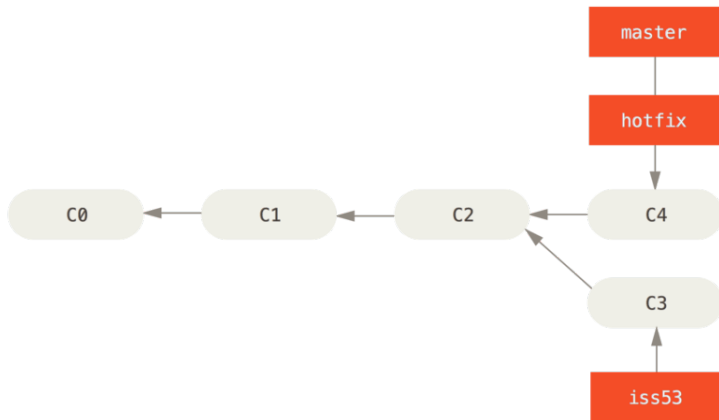
Merging Branches

- ▶ In this case, the merge is a "fast forward"
- ▶ This is because the hotfix pointer was directly ahead of master
- ▶ So the master pointer can simply be moved forward to hotfix

```
sc@sc-ubuntu:~/MyRepo$ git checkout master
Switched to branch 'master'
sc@sc-ubuntu:~/MyRepo$ git merge hotfix
Updating a6ef6ac..9c1f2ee
Fast-forward
 README | 1 +
 1 file changed, 1 insertion(+)
sc@sc-ubuntu:~/MyRepo$
```

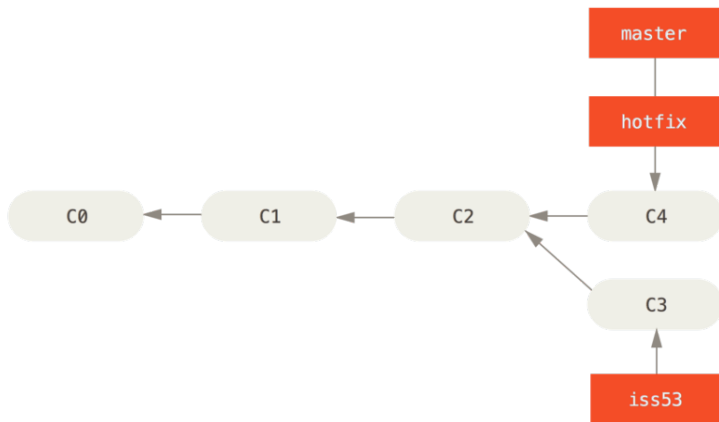
Merging Branches

- ▶ In this case, the merge is a "fast forward"
- ▶ This is because the hotfix pointer was directly ahead of master
- ▶ So the master pointer can simply be moved forward to hotfix



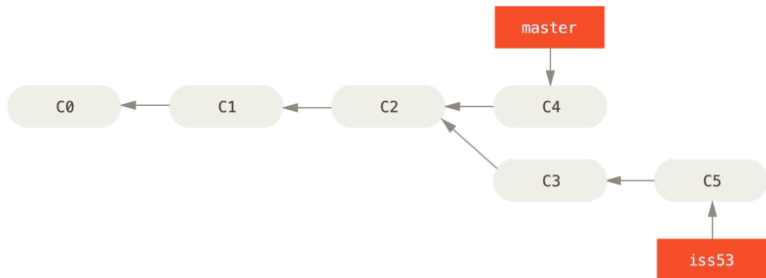
Merging Branches

- ▶ Once a branch has been finally merged you can delete it
- ▶ Do this with **git branch -d hotfix**
- ▶ This only deletes the pointer, the commits are safely on master



Merging Branches

- Now we switch back to iss53 branch and continue committing



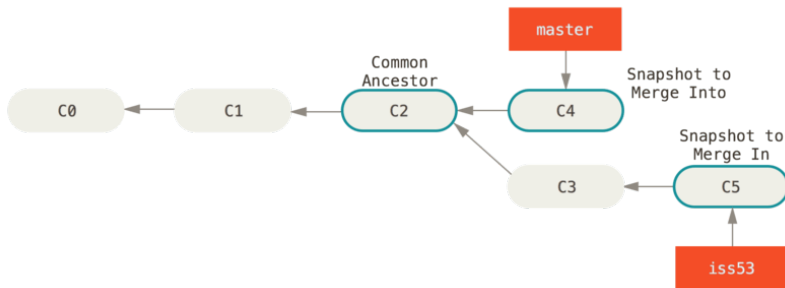
Merging Branches

- Once iss53 is complete, it can be merged into master

```
sc@sc-ubuntu:~/MyRepo$ git checkout master
Switched to branch 'master'
sc@sc-ubuntu:~/MyRepo$ git merge iss53
Merge made by the 'recursive' strategy.
 new.c | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new.c
sc@sc-ubuntu:~/MyRepo$
```

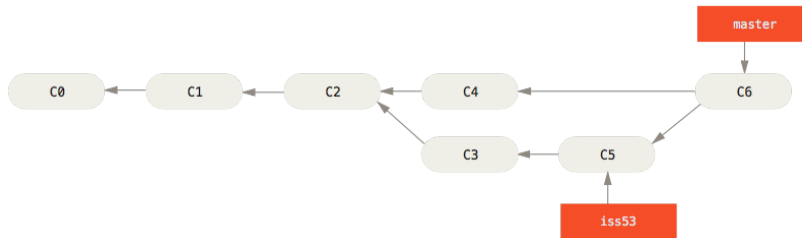
Merging Branches

- ▶ This merge wasn't a fast forward
- ▶ This is because the current commit in master isn't a direct ancestor of the top commit in iss53



Merging Branches

- ▶ Git creates a new snapshot for this merge
- ▶ A new commit is create that points to it
- ▶ This is often called a "merge commit"
- ▶ This merge commit has two parents



Merge Conflicts

- ▶ Not all merges go so smoothly
- ▶ If the top snapshot on each branch has a different version of the same file a merge conflict occurs

```
sc@sc-ubuntu:~/MyRepo$ git merge iss53
Auto-merging new.c
CONFLICT (content): Merge conflict in new.c
Automatic merge failed; fix conflicts and then commit the result.
sc@sc-ubuntu:~/MyRepo$
```

Merge Conflicts

- ▶ Not all merges go so smoothly
- ▶ If the top snapshot on each branch have different versions of the same file a merge conflict occurs

```
sc@sc-ubuntu:~/MyRepo$ 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:   new.c

no changes added to commit (use "git add" and/or "git commit -a")
sc@sc-ubuntu:~/MyRepo$
```

Merge Conflicts

- ▶ To resolve, open the conflicting file(s) in your editor
- ▶ Manually resolve the conflicts
- ▶ Git labels the conflicting segments of the file
- ▶ In this case, pick one version and delete the other

```
1 <<<<<< HEAD
2 #include <stdlib.h>
3
4 int main(int argc, char **argv)
5 {
6     return EXIT_SUCCESS;
7 =====
8 int main()
9 {
10     return 0;
11 >>>>>> iss53
12 }
```

Merge Conflicts

- ▶ Once the files have been edited to resolve the conflicts:
- ▶ Stage the files
- ▶ **git commit**

```
sc@sc-ubuntu:~/MyRepo$ git add new.c
sc@sc-ubuntu:~/MyRepo$ git status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

nothing to commit, working directory clean
sc@sc-ubuntu:~/MyRepo$ git commit
[master 9244d59] Merge branch 'iss53'
sc@sc-ubuntu:~/MyRepo$
```

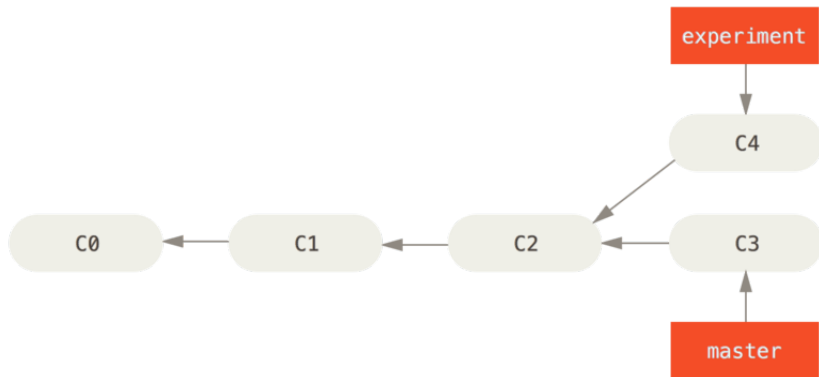
Branch Management

- ▶ List branches with top commit on each: **git branch -v**
- ▶ List branches that are merged into current branch: **git branch --merged**
- ▶ List unmerged branches: **git branch --no-merged**

Rebasing

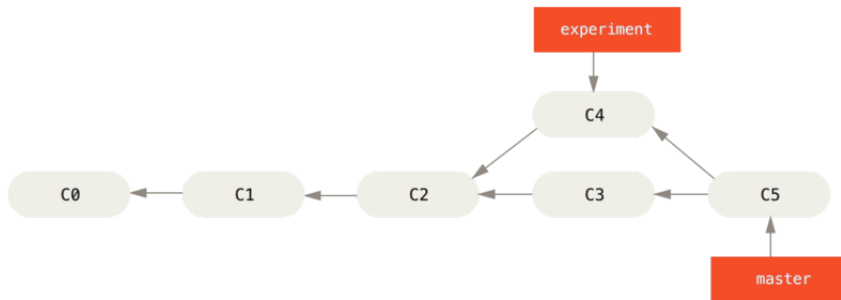
- ▶ Rebasing is another way of integrating changes from one branch into another
- ▶ In some situations it's better than merging, in some it's worse, and in some you shouldn't do it at all

Rebasing



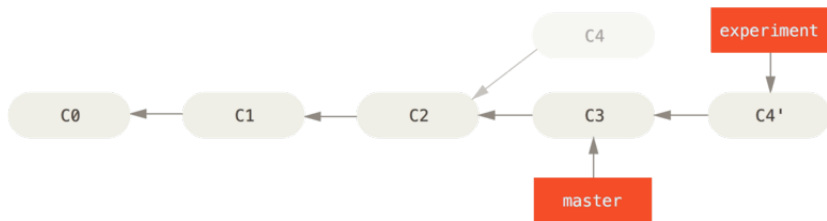
Rebasing

- ▶ Two branches merged normally



Rebasing

- ▶ Can use rebasing to keep the history linear
- ▶ Take the patch introduced in C4 and reapply it on top of C3
- ▶ Avoids a merge commit



Rebasing

- ▶ Can use rebasing to keep the history linear
- ▶ Take the patch introduced in C4 and reapply it on top of C3
- ▶ Avoids a merge commit

```
sc@sc-ubuntu:~/MyRepo$ git log --all --decorate --oneline --graph
* 3ddc097 (master) Add CONTRIBUTING
| * 1aec4b9 (HEAD -> experiment) Add LICENCE
|/
* e1f1735 Add README
sc@sc-ubuntu:~/MyRepo$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: Add LICENCE
sc@sc-ubuntu:~/MyRepo$ git log --all --decorate --oneline --graph
* 3b0ea45 (HEAD -> experiment) Add LICENCE
* 3ddc097 (master) Add CONTRIBUTING
* e1f1735 Add README
sc@sc-ubuntu:~/MyRepo$
```

Rebasing

- ▶ Now the history has been linearised with regard to the two branches
- ▶ Just need to update master's branch pointer

```
sc@sc-ubuntu:~/MyRepo$ git checkout master
Switched to branch 'master'
sc@sc-ubuntu:~/MyRepo$ git merge experiment
Updating 3ddc097..3b0ea45
Fast-forward
 LICENCE | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 LICENCE
sc@sc-ubuntu:~/MyRepo$ git log --all --decorate --oneline --graph
* 3b0ea45 (HEAD -> master, experiment) Add LICENCE
* 3ddc097 Add CONTRIBUTING
* e1f1735 Add README
sc@sc-ubuntu:~/MyRepo$
```

Rebasing

- ▶ Now the history has been linearised with regard to the two branches
- ▶ Just need to update master's branch pointer
- ▶ Due to the rebase, the merge is a fast-forward, hence no commit

