Embedded Systems Programming Laboratory

Git

Alex Hoffman

Technical University of Munich
Chair of Real-Time Computer Systems (RCS)

October 10, 2022

Lehrstuhl für
Realzeit-Computersysteme

What is Git?

- Developed in 2005 by Linus Torvalds
- Git is a versions-control system for tracking the changed made to a codebase.
- Works in a distributed fashion - every machine has a copy of the codebase.
- The industry standard on version-control
- A core skill you will need if you plan to work in a job where you interact with code in any manner

Basics

Repositories (repos):

- Mini filesystem that is seen as a folder on your system
- All changes to the contents of this folder is tracked
- Changes are stored using snapshots at user defined moments in time (commits, more on this later)
- Each moment in the repo's history is given a unique value that allows the user to move between these moments
- Remote server stores a copy of the repository that acts as a centralized master version
- Configuration stored in the `.git/config` within the repo

Creating a repository

Usually done on the web interface of the back end you wish to use. Github, Gitlab etc. Note, Github profiles are becoming increasingly important for showcasing your coding abilities to employers etc.

- In the web interface click on create new repository or similarly named function
- Name the repository and select appropriate starting contents
  - README.txt - gives an overview of and/or instructions for the project. Web interface's page's contents generated from README files.
  - Licence - defines the licensing of the project's contents.
  - .gitignore - explained later
- Either clone the created repository or initialize the repository in an existing folder
  - `git clone` - Creates a copy of the repository (with appropriate config files) locally on your machine
  - `git init` - Creates empty git config files in an existing folder, the server (remote) must be then added to the repository using `git remote add`.

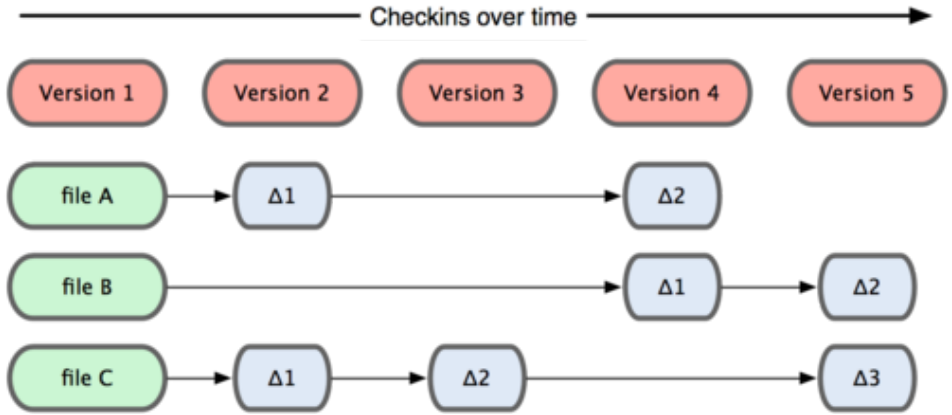
Creating checkpoints in your folders - Commiting

Commits:

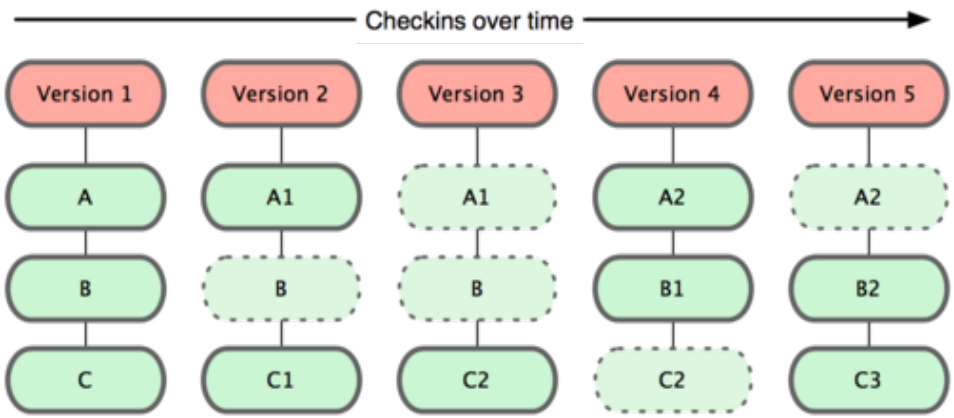
- Hold a snapshot of the Git directories state
- Snapshot is comprised of copies of changed files and references to existing copy of unchanged files. Stored in a compressed "blob".
- Commits are referenced by unique SHA-1 hashes generated from the commit's data
- All commits must contain a commit message
- `git commit -m "commit message here"`

Git vs other Version Control Software (VCS)

VCS



Git



https://git-scm.com/book/en/v1/Getting-Started-Git-Basics#Snapshots,-Not-Differences

Commit messages - best practise

- Should not explain HOW something changed, this can be seen by `diff`.
  - example: "Created ButtonPolling class and added private button handling functions"
- Should explain WHAT and WHY something was changed, give context!
  - example: "Button polling module added to consolidate button handling into a stand along interface"
- 7 rules for a great commit message:
  - Separate subject from body with a blank line
  - Limit the subject line to 50 characters
  - Capitalize the subject line
  - Do not end the subject line with a period
  - Use the imperative mood (leave our unnecessary words) in the subject line
  - Wrap the body at 72 characters
  - Use the body to explain what and why vs. how

Commit message example

```
Summarize changes in around 50 characters or less

More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of the commit and the rest of the text as the body. The
blank line separating the summary from the body is critical (unless
you omit the body entirely); various tools like ‘log‘, ‘shortlog‘
and ‘rebase‘ can get confused if you run the two together.

Explain the problem that this commit is solving. Focus on why you
are making this change as opposed to how (the code explains that).
Are there side effects or other unintuitive consequences of this
change? Here’s the place to explain them.

Further paragraphs come after blank lines.

 - Bullet points are okay, too

 - Typically a hyphen or asterisk is used for the bullet, preceded
   by a single space, with blank lines in between, but conventions
   vary here

If you use an issue tracker, put references to them at the bottom,
like this:

Resolves: #123
See also: #456, #789
```

Which files are commited?

The Git workflow is slightly different from other VCS

1. Working directory - the aforementioned mini file system which Git implements
   - Here files are created/modified/deleted
2. Staging area
   - The user informs Git which files are important to the commit
   - Files of interest are added to the staging area, thereby "tracking" them
   - Tracked files are included into the "snapshot" created by a commit
   - Files are added to the staging area using the `git add` command
   - Difference between two commits is seen as a set of changes/patch
   - Allows for changes to be put into separate commits by controlling which changes are explicitly in the commit, allows for easy patch generation

Which files are commited?

3. Repository - the remote server storing the central copy of the code base
   - Once files commited from the staging area into a commit they must be pushed to the remote repository
   - Done using `git push`
   - `git push` specifies the remote server and branch that should be push to and what should be push. Remote aliases are specified in the Git config file. Branches will be detailed shortly.

Example:

```
> echo "Hello world" > new_file.txt

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

  new_file.txt

> git add new_file.txt
```

Commit example cont.

```
> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  new file:   new_file.txt

> git commit -m "Creating a new file to show how commiting
    works"
[master 69798c1] Creating a new file to show how commiting
    works
 3 files changed, 31 insertions(+), 1 deletion(-)
 create mode 100644 new_file.txt

> git push origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 237 bytes | 237.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0)
```

Branches

Benefits:

- Allow for parallel development of code
- Allows for multiple versions of code to exist, ie. a "stable" copy and a "unstable" development copy
- `master` branch is the default core branch of any repository and should be treated as sacred
- Do not develop on `master`
- Branches enable `git merge` which I will outline next

Mechanism:

- Branching, as the name implies, branches the current commit into a separate branch
- Future commits are commited to the current branch and not the branch that was branched from

Branching - best practise

- Master or production branches should be held in a stable state at all times (not developed on)
- Features or fixes are developed in separate branches.
- Completed features are merged into master or production branches before being pushed to the remote

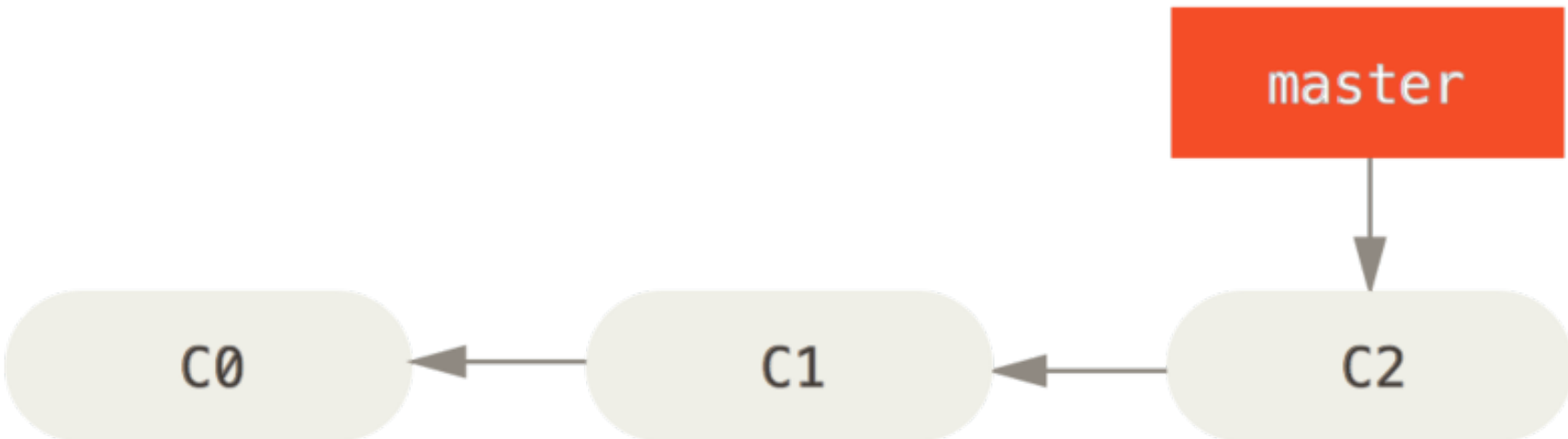
Commands:

- `git branch branch_name` - Creates a branch with the name `branch_name`
- `git checkout branch_name` - Checkouts the branch `branch_name`, done so by setting the `HEAD` pointer at the specified branch
- `git checkout -b branch_name` - Creates and checksout the branch `branch_name`

`HEAD`: A reference variable that stores the location of the most recent commit of the current branch.

Branching - visual example 1

A branch with 3 existing commits

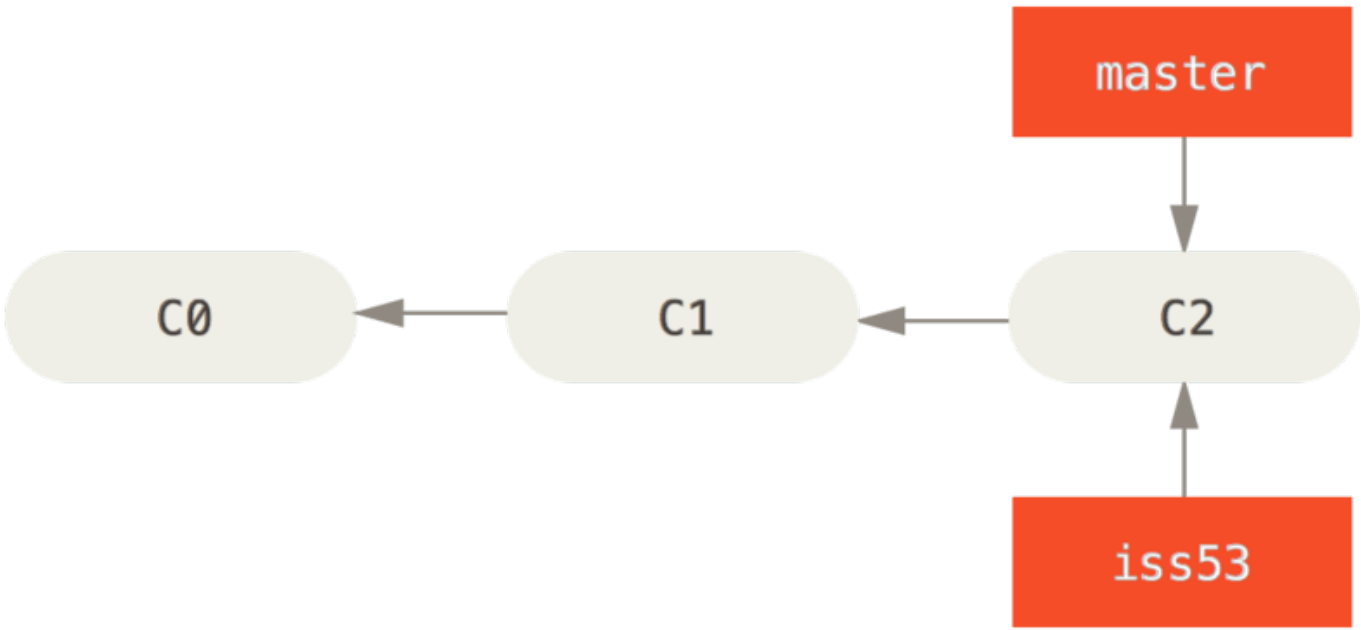


https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Branching - visual example 2

A branch called iss53 is created, its `HEAD` is set at C2

```
$ git checkout -b iss53
```

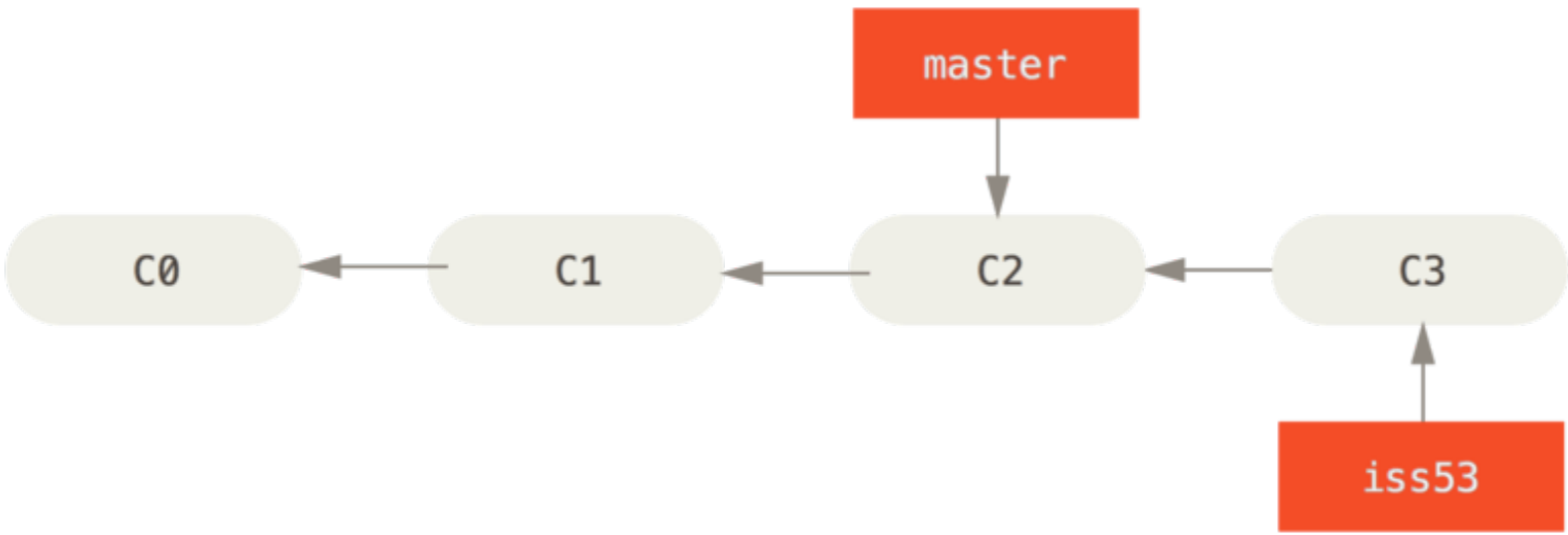


https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Branching - visual example 3

Commit C3 is created on the iss53 branch

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

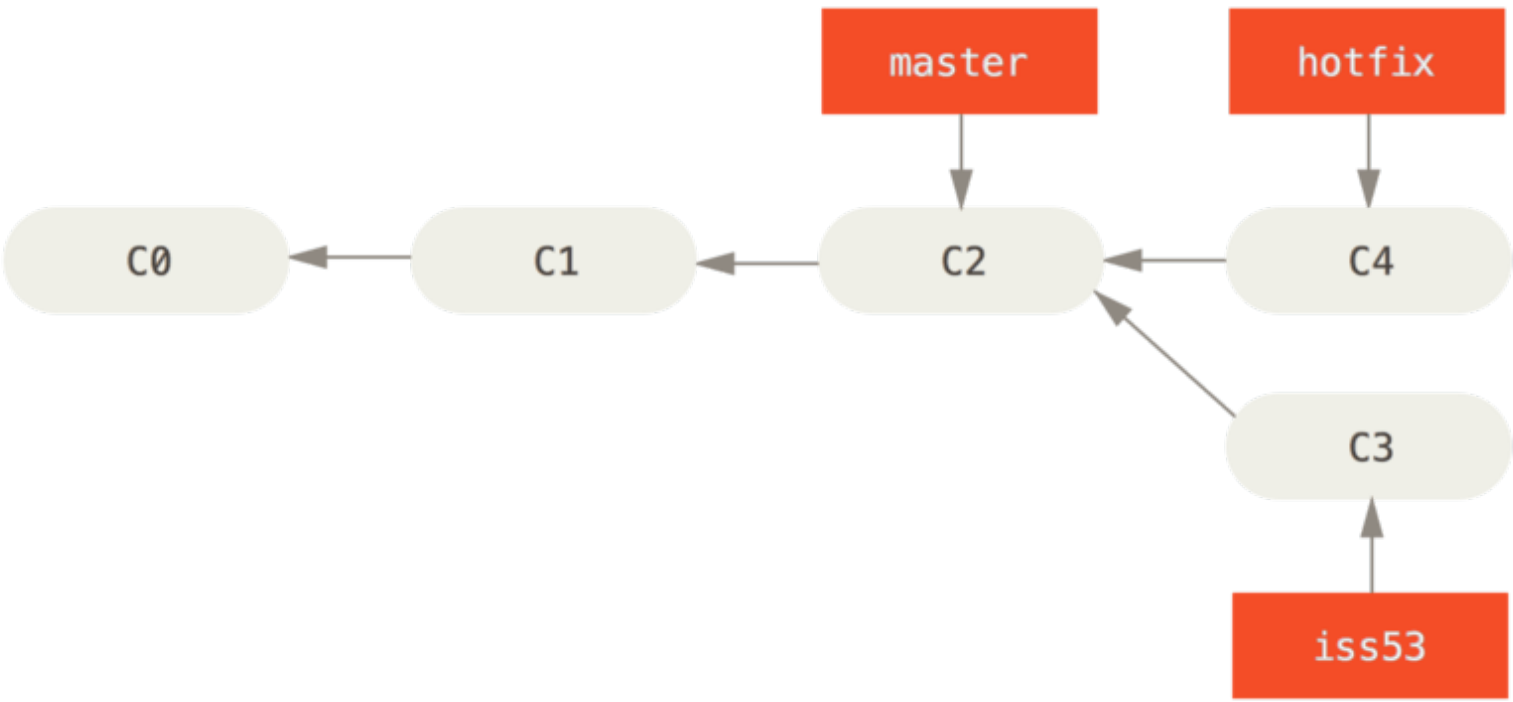


https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Merging

For the sake of example, lets say you created another branch to fix another issue, seen below

```
$ git checkout -b hotfix
... doing some changes and commiting them
```



https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Merging

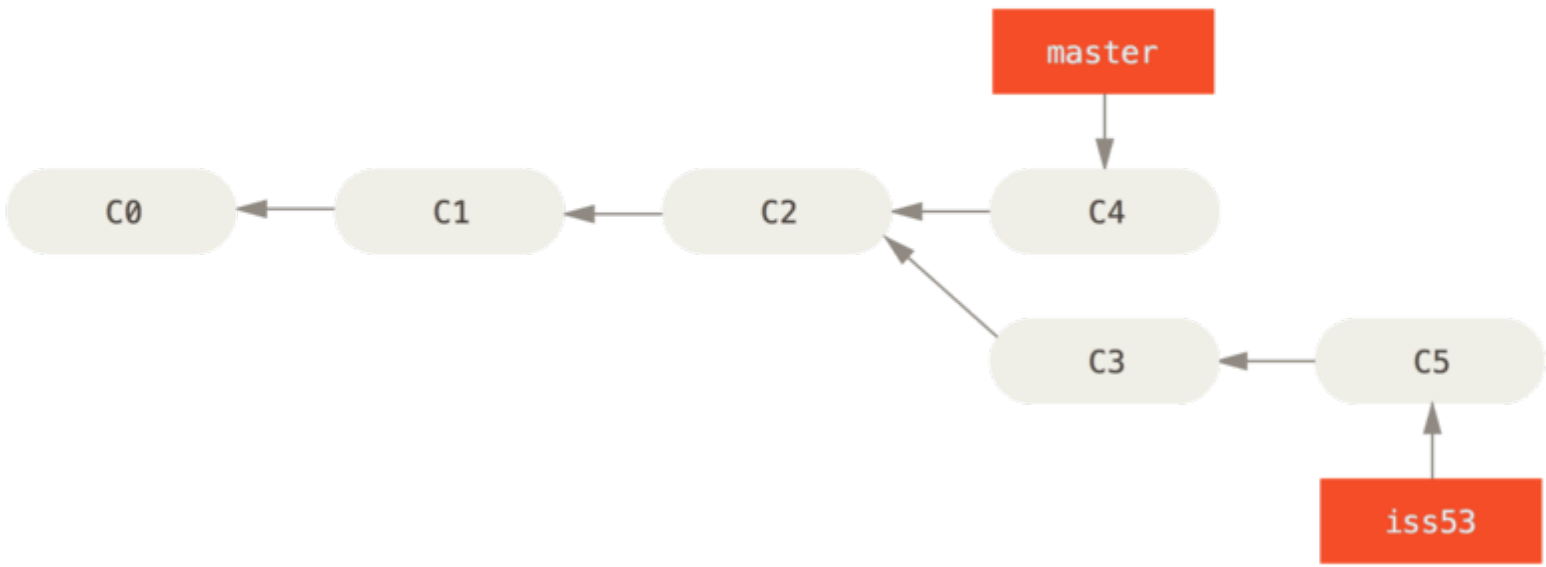
- Merging is the act of integrating the code bases of two branches together
- The most basic merge (a "fast-forward" merge) is simply the moving of the HEAD pointer to point to the most recent commit in the branch being merged
- Git can handle most code merging automatically but human intervention is required sometimes
- Merge conflicts occur when branches has diverged from a common ancestor

Commands:

- `git merge branch_name` - Performed from the branch into which you wish to merge the branch `branch_name`

Merging example

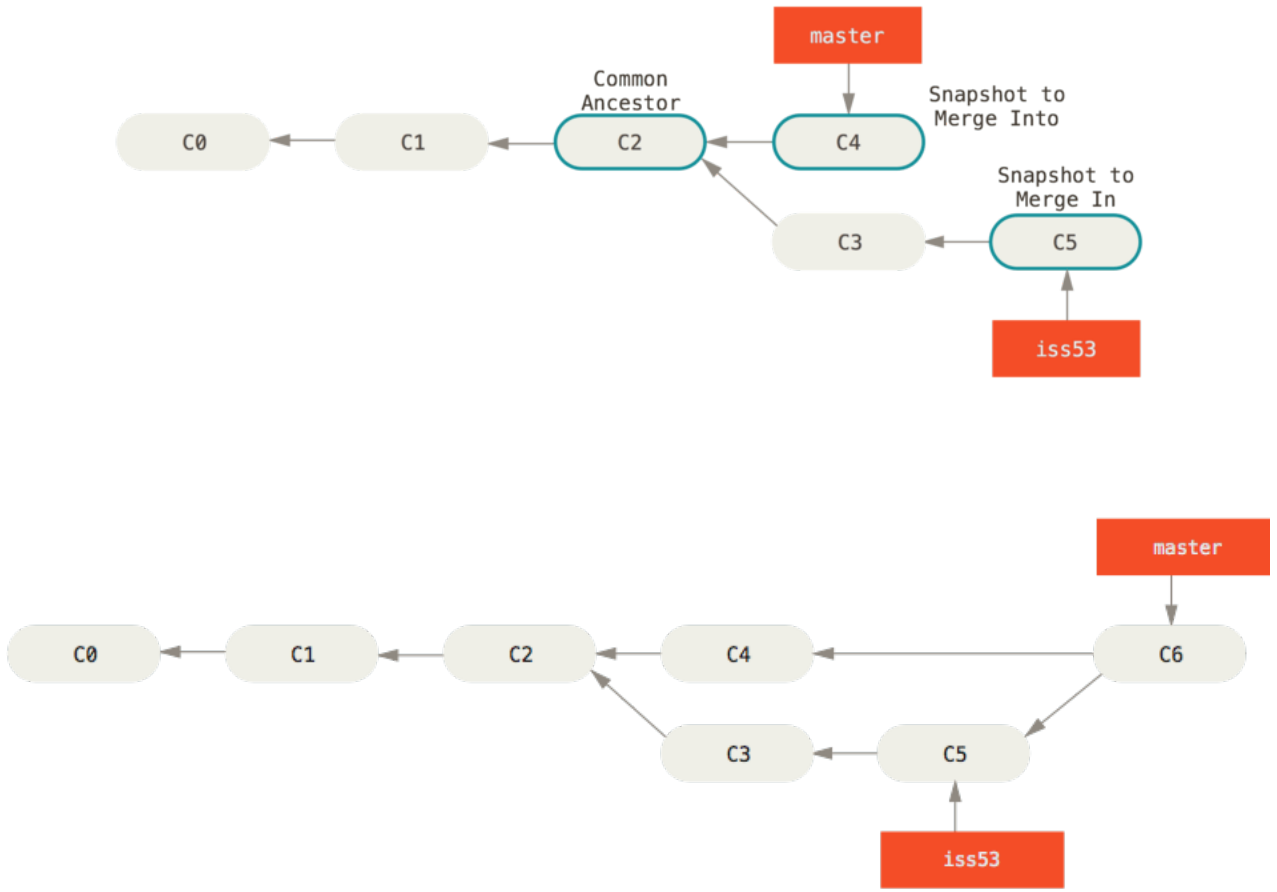
Merging `hotfix` into master moves the master's HEAD pointer to C4



https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Merging example cont.

Merging `iss53` into master then creates a new commit, termed merge commit, into which the two branches are merged.



https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Merge conflicts scenario

Merging two branches that have diverged from a common ancestor can cause merge conflicts.

Example scenario:

- Developer 1 and developer 2 both branched from the same commit.
- Developer 1 renames the function `initsystem` to `initSystem`
- Developer 2 rename the same function to `init_system`
- Developer 1 merges their changes into master which fast-forwards master
- Developer 2 merges their changes into master causing by performign a three way merge with the two commit and the common ancestor
- A merge conflict occurs as the branches diverged from a common ancestor and Git cannot know which change is the correct one

Merge conflicts scenario cont.

Git would notify the user merging of the conflict.

```
$ git merge snake_case
Auto-merging main.c
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the
    result.
```

- The merge process has been paused, no new commit has been generated as the conflict must be resolved first.
- Git adds standard conflict-resolution markers

Merge conflicts scenario cont.

```
<<<<<<< HEAD
void initSystem( void )
=======
void init_system ( void )
>>>>>>> snake_case
```

The conflict markers show that HEAD contains the line `initSystem` while the `snake_case` branch's HEAD contains the line `init_system`.

Resolving this must be done by hand or with the aid of a merge tool. The files must then be added to the commit that is committed as a result of the resolved merge conflict.

Other useful commands

- `git pull`
- `git fetch`
- `git stash`
- `git show`
- `git revert`
- `git clean`
- `git rebase`
- `git cherry-pick`

Git Pull and Git Fetch

Example scenario:

- A colleague of yours has implemented some new features, merging and pushing them into the master branch
- You are currently working on some other feature that will interact with the features they have developed so you must make sure they are compatible

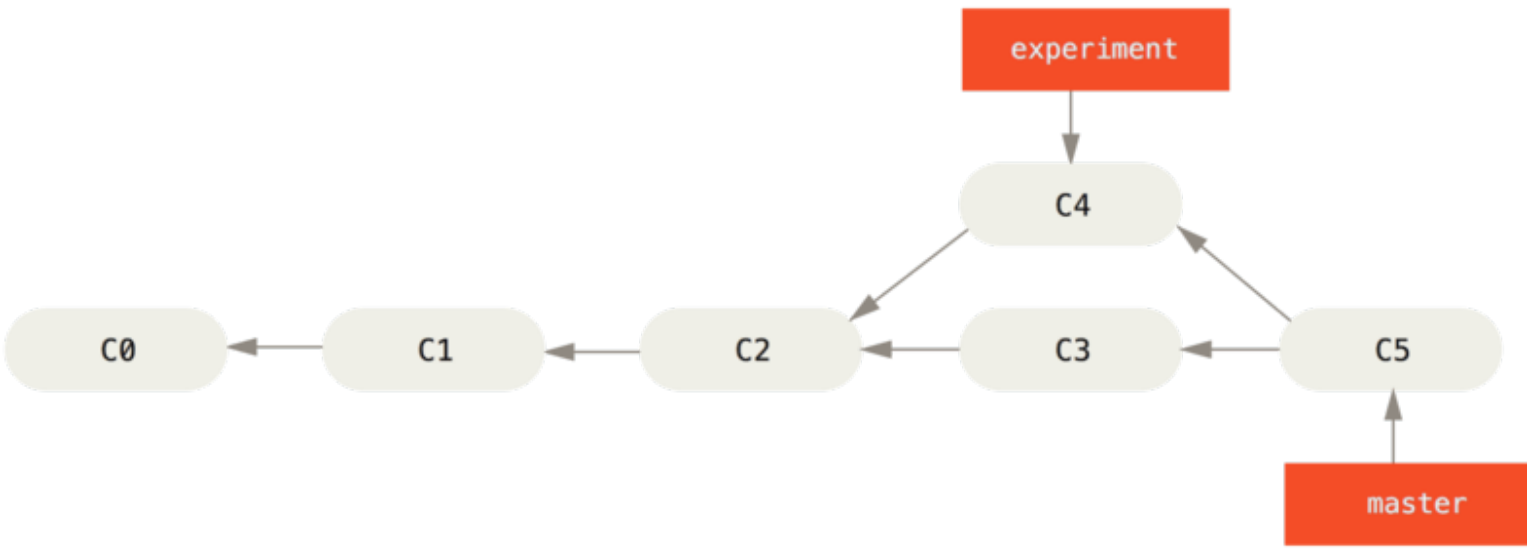
Fetch: *Downloads all the data from a specified remote that you don't have yet. Allowing you to inspect all new branches (that you now have references to) or merge new branches into your branches.*

Pull: *Downloads and merges all the data from the remote for the branch you are currently tracking.*

Git Rebase

Example scenario:

- The same happened as previously mentioned in the merge scenario
- Instead of performing a three-way merge to merge the two diverged branches one can create a patch from one branch and apply it to the other diverged branch
- Thus all the changes from one branch are taken and applied to another branch

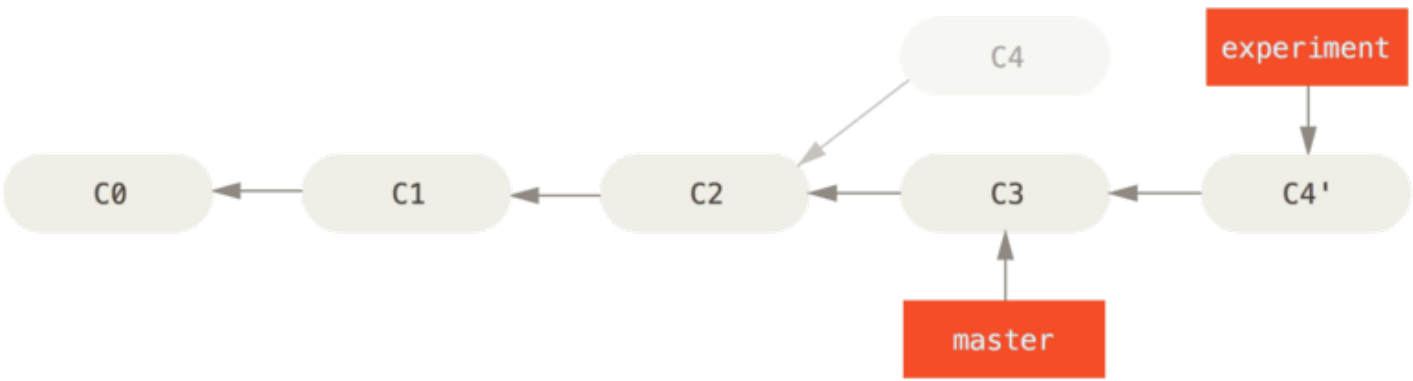


https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Git Rebase Cont.

Steps performed:

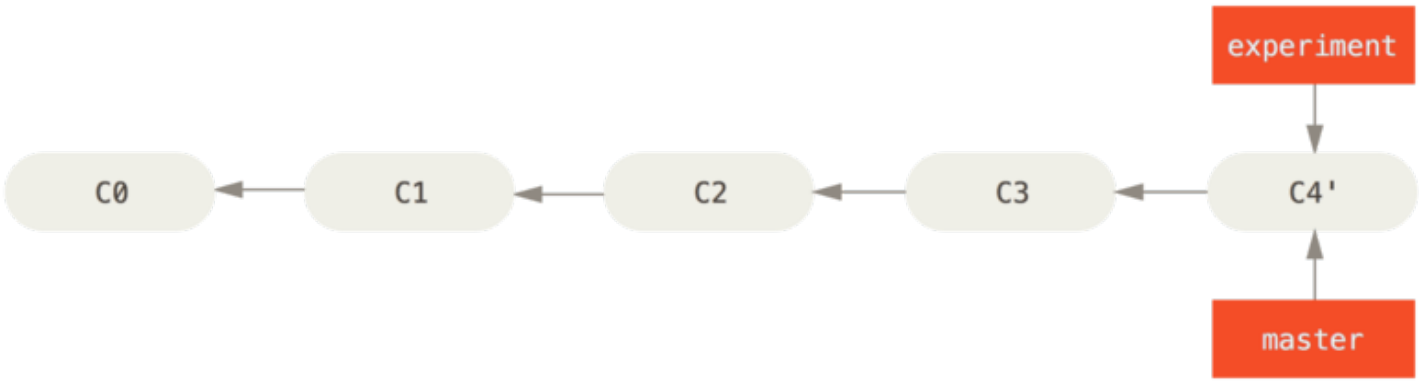
- From the common ancestor node all commit diffs are saved into temporary patches
- Current branch is reset to the same commit from the branch into which you're merging
- Each patch is applied in turn



https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Git Rebase Cont.

Fast-forwarding master



https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

The end result is the same as merging although the Git history is cleaner as as the work appears to happen in series instead of in parallel.

Git Stash

Use case: If you are working on a particular feature, but it is not ready for merging or publication, and you wish to change branches and work on something else for a bit then you can save the current state of your repository and return later without commit incomplete work.

Features:

- Uses of a stack to store changes
- Returns your code base to a clean state (last commit in branch)

Git Stash Cont.

Commands:

- `git stash push` - pushes a new stash onto the stack
- `git stash list` - lists the stored stashes
- `git stash apply` - applies a previously saved stash
- `git stash drop` - removes the named stash from the stack
- `git stash pop` - applies the top stash on the stack and drops the stash

Behaviour:

- `stash@{0}` - top stash of stash stack and default stash if none is given when using `apply`
- Can be applied to a dirty directory or another branch, behaved like a merge and can result in conflicts
- Untracked files can be added to a stash by using the `-u/-include-untracked` option

Git Clean

Warning: can cause a lot of harm if used improperly

- Used to remove unwanted untracked files from your Git directory.
- Useful for removing build artefacts that your gitignore specify
- `git stash -all` - a safe version where a mistake execution can be retrieved again

Git Tutorial

- Find me on GitHub (https://github.com/alxhoff)
- Find my pinned repository "git-tutorial"
- **FORK!!!** the repository (not clone)
- Follow instructions on website (in README)
- It's a lot of reading and a lot to take in, but it is full of VERY valuable industry knowledge/skills

Sources

www.freertos.org

http://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

http://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V9.0.0.pdf