Version Control System - Git

A lot of people work on developing features, fixing bugs, testing, deploying and releasing from same source code at the same time. So source control software are integral part of the software development. These software maintain the source code and keep it accessible for anyone at anytime. Several version control software (VCS) have been popular in the past. These days more than 70% of developers use git.

A. Introduction

Git is open-source software. It is distributed version control system which means that the same source code can be present on several servers (in the form of remote repositories) and local machines (in form of local repositories). These repositories can be updated on the go from another repository. In other VCSs, access to central server was needed at all times no matter how small access is needed. But in Git almost all the operations can be performed locally and data can be uploaded once the server is available. (An employee can work on tickets on local machine and upload the work when he gets access to the company's network - VPN.)

Git uses the SHA-1 hash function to name content. For example, files, directories, and revisions are referred by hash values unlike in other traditional VCSs where files or versions are referred via sequential numbers. The use of a hash function to address its content delivers a few advantages:

- Integrity checking is easy. Bit flips, for example, are easily detected, as the hash of corrupted content does not match its name.
- Lookup of objects is fast.

Git stores each file in form of objects in its database and then maintains pointers of the objects in hierarchy. These objects are?present in .git folder inside each repository. Git maintains this folder itself.

B. Model

Git maintains the pointers to the object in each state. In this way the pointer to object in one snapshot is only changed when the object is changed.

Assume we have three files which can be changed along the way. As the file is changed in the next snapshot, only pointer to that file is changed and pointer to other files remains same. Same files aren?t copied between snapshots. Thus a lot of space is saved and speed is achieved.

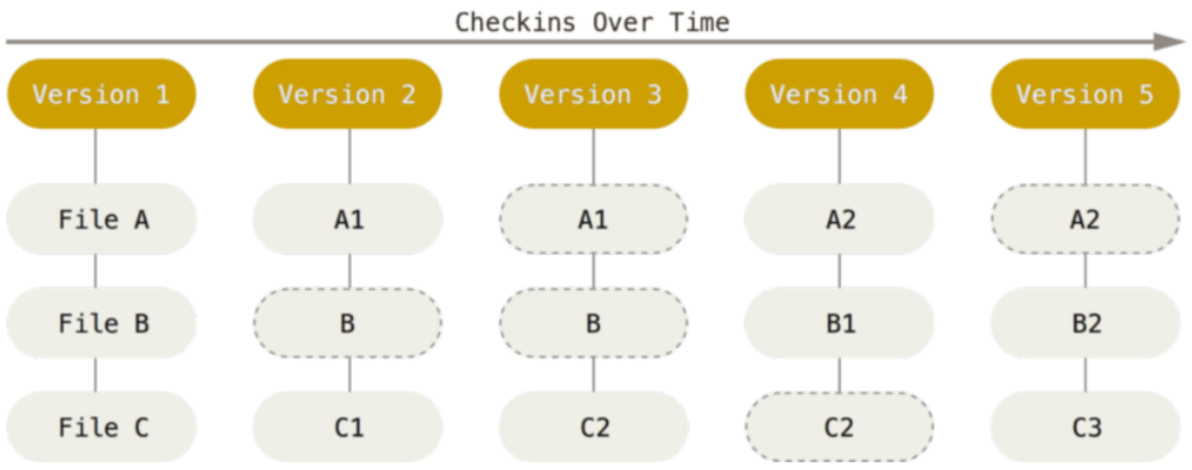
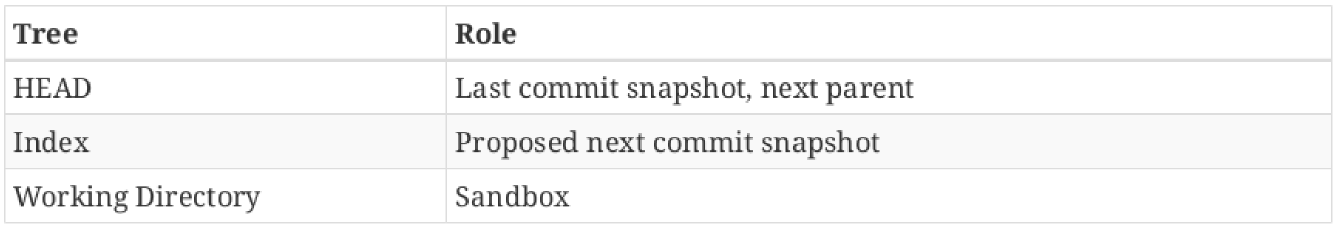


Figure 5. Storing data as snapshots of the project over time.

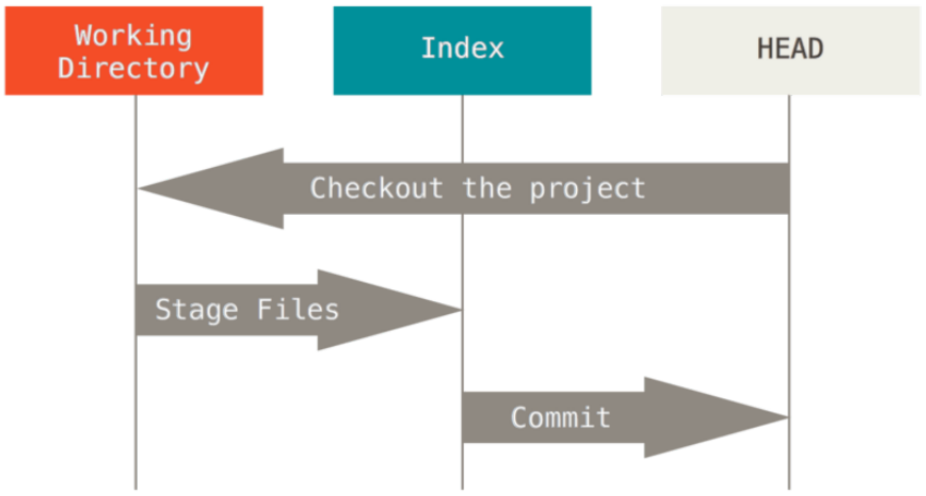
The Three Trees

Git has three trees based model. (These trees represent the high level level usage point of view. Local repository should be considered divided in these trees.)



Tree	Role
HEAD	Last commit snapshot, next parent
Index	Proposed next commit snapshot
Working Directory	Sandbox

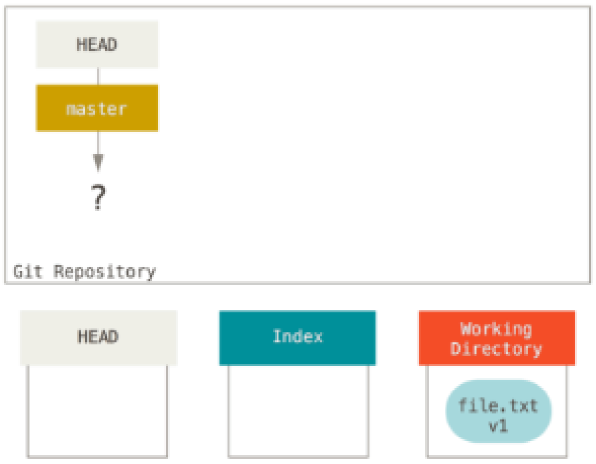
If changes are made to a file, these changes happen in the ?Working Directory? tree. If these changes are staged, the changes go to the "Index" or "staging" tree. Then if a commit is made, the changes are permanently stored as a individual entity. It is called HEAD. Any source code checkout happens from "HEAD" tree.



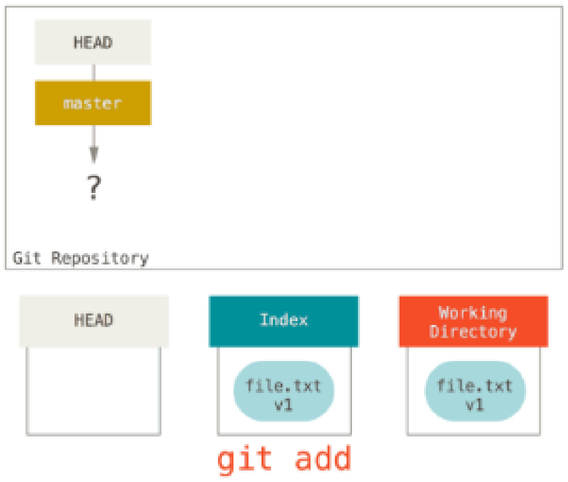
Understanding the tree with example

Only understand the concept in the following example. Commands to perform those operations will be explained later.

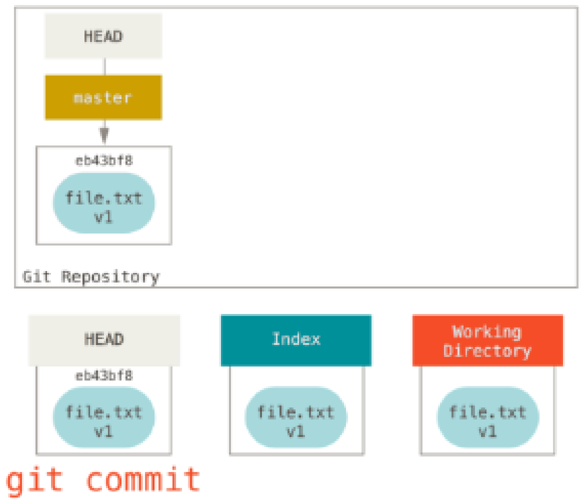
Suppose there is a empty repository. You have copied a file in that repository. This file is contained in the working directory at this time.



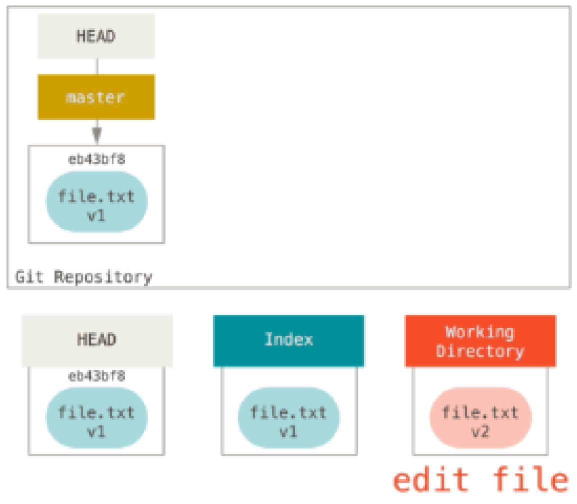
Now this file has been added to the index. After adding to index, git will start tracking the changes in this file.



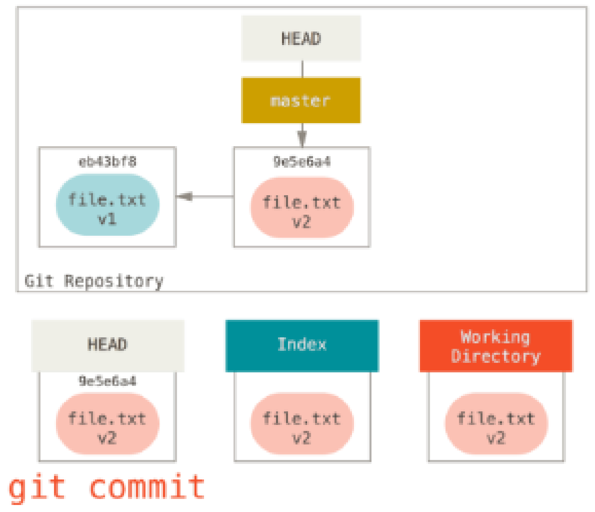
If at that point a commit is done, the changes of this file will be saved in the git?s history. This new commit will be visible at the HEAD(case sensitive) also.



If some changes are made to this file, it?ll look like this figure. The changes will remain in the working directory until those changes are added to Index tree.



So the v2 file has been added and then commited. HEAD has another commit and everything is in the hierarchy. Second commit contains just the diff between two versions.



Note that this *file.txt* has history in both of the commits now.

C. Basic Commands

Now we'll learn git by hands on experience. Please install git and git-lfs on your machine before going forward from here.

Create and clone the first repository

Beginners should follow the first method described below:

1. Usually repositories are created on a server (Stash, BitBucket or Github) and then cloned in the local machine for use.
   a. Open https://stash.alm.mentorg.com/profile and select "Create Repository".
   b. Add name and description of the repository. Click on "Create Repository" and you are done.
   c. Now the repository can be cloned on the local machine using the following command:
      ```
      git clone https://stash.alm.mentorg.com/scm/~manjum/hello-git.git
      ```

2. A local repository can also be created and then sent to the server after creating the repository on the server later on.
   a. Create a directory and go into it from the shell and run the following command to create a git repository:
      ```
      git init
      ```
   b. Add a remote of this repository (First create this repository on server with the method described above.)
      ```
      git remote add origin?https://stash.alm.mentorg.com/scm/~manjum/hello-git.git
      ```
   c. Do a commit in this repository. (See First time configurations and First commit sections below)
   d. Now we can push to the remote repository
      git push --set-upstream origin master

First time configurations

Git records the name and the email of the developer in every commit. So run the following commands to set the name and email before doing any commit etc.

```
git config --global user.name "Anjum, Muhammad Usama"
git config --global user.email "MuhammadUsama_Anjum@mentor.com"
```

First Commit

The `git commit` command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as ?safe? versions of a project. Let's do the first commit in the repository which we have created above:

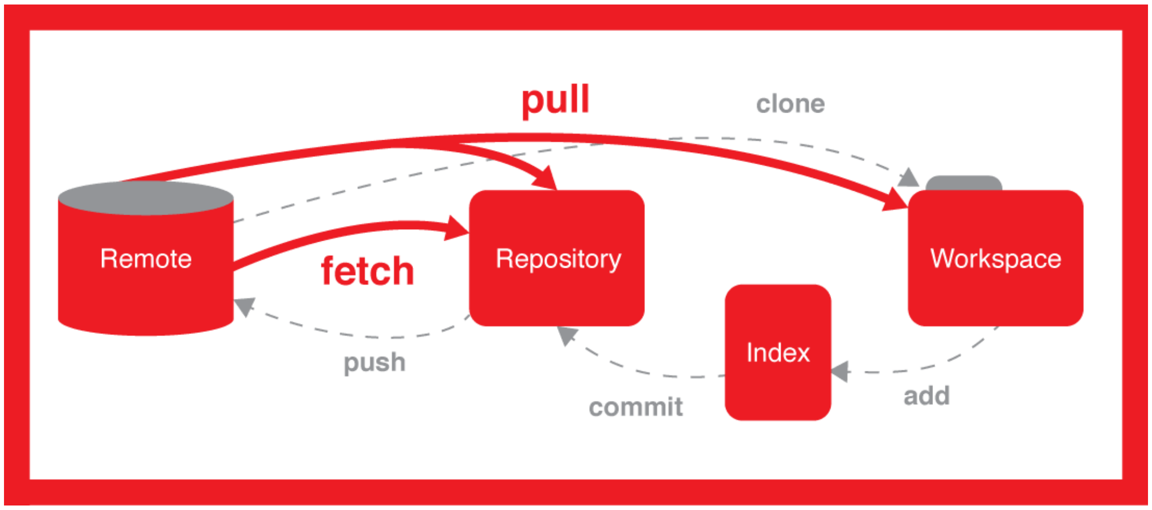
- Add a file in the root of the repository with name hello.c and with the following contents:
  ```
  #include <stdio.h>

  int main()
  {
  ??? printf("Hello world\n");

  ??? return 0;
  }
  ```
- This file has been created in the "working directory" tree. The following command can be run to find this out also:
  ```
  git status
  ```
- Now let's stage this file:
  ```
  git add hello.c
  ```
- Do the commit:
  git commit -m "Add the hello.c file"
- To send this commit to the remote push this current state to the remote repository:
  git push

Note that git push is always used to push the current branch changes to the remote repository. In this case the our first commit will be sent to the remote repository.

Summary:

The commit we have done above is being done in the "master" branch. This "master" branch is the most significant main branch. A lot of people keep on updating this branch all the time. So before we commit anything we should update our local repository to the latest that of remote branch in remote repository. It should be noted that the remote repository is higher priority repository. So if there is unknown commit in the "master" branch, always consider that some other developer must have done that.

So one repository is remote and one repository is local. If we want to only bring changes from remote repository to the local repository, we need to run git fetch only. Git fetch will not update the "working tree" until we use git pull. Note that git pull performs git fetch also under the hood. The following figure should be 100% clear at this point. (If not, please ask your trainer to explain it to you more.)



Practice git fetch/git pull:

- Clone the same repository second time into a new directory "hello-git1"
  git clone https://stash.alm.mentorg.com/scm/~manjum/hello-git.git hello-git1
- Once this same repository is cloned into a new place, you should find the commit you had done above by running:
  git log
- Let's update the hello.c file. Add the following include on the second line:
  #include <stdlib.h>
- Run git status now to find out that this hello.c file has been changed in the "working directory" tree. Let's stage it:
  git add hello.c
- Let's do the commit:
  git commit -m "Add the stdlib header file"
- Let's push it to the remote
  git push
- Now go back to the previous first cloned repository named as "hello-git" and run git status. You'll know that this repository doesn't even know that the remote repository has been updated and moved on.
- At this point we need to run git fetch or git pull to update this local repository with the remote repository.
  - If we do git fetch and then do git status, now the status will tell us that the remote repository was updated and we can update our local copy to that one by running the git pull. So do git pull at this point.
  - If we do git pull only, it'll update the local repository and also update the current working directory tree for us.

Now you should understand this git fetch/pull properly. (If not, please practice more.)

Git add explained:

Whenever we do a commit, the changes which are staged (index-ed) become the commit. To move changes from working directory to the "Index" tree or "staging area", git add command is used. Git add can be used to add one file at a time or we can give it a directory name and it'll add all the files changed in it recursively. Suppose we have 5 files in a directory "abcd" in our git repository.

- Add each file one by one:
  git add abcd/file-1.c
  git add abcd/file-2.c
  git add abcd/file-3.c
  git add abcd/file-4.c
  git add abcd/file-5.c
- Add all files inside that directory with one command:
  git add abcd

To unstage (remove file from index) a file and bring it to just working directory tree, following command can be used:

git rm --cached file.c

Git commit explained:

After staging the files we can do the commit in one of the following ways:

- Run:
  git commit
  A editor will open for you to write the commit message. Write the informative commit message and close the editor.
- Run the following command:
  git commit -m "informative commit message"

Note that some people use git commit -a -m "message" command to stage all the changed files and then commit with one command. But this method is highly discouraged until you know what you are doing as there may be certain changed files which you don't want to commit.

Git status explained:

Git status is used to find the current status of the repository at any time. This command can be used with several other flags for different kinds of statuses. Try running the following command to checkout one variant of this command for now:

git status -s

Git log explained:

Git log is used to find out the history of commits in hierarchy. For example on our created repository if we run git log, it'll show us the 2 commits we have done in order, with commit messages and some more information. Try out some other variants on the git log command:

git log --pretty=oneline

git log --since=2.weeks

Git blame explained:

Git blame is used to check the blame history of a file. It shows who had changed which line of the code through which commit.

git blame <file path>

D. Branching Model and Commands

Branching means you diverge from the main line of development and continue to do work without messing with that main line. Some people refer to Git?s branching model as its ?killer feature,? and it certainly sets Git apart. The way Git branches is incredibly lightweight, making branching operations nearly instantaneous, and switching back and forth between branches generally just as fast. Git encourages workflows that branch and merge often, even multiple times in a day. Understanding and mastering this feature gives you a powerful and unique tool and can entirely change the way that you develop. The default branch name in Git is `master`. Usually?`master` is the default branch and no developer can merge directly to this branch (until you own the repository and want to do it). People create branches from?`master` branch and then merge them into the `master`.

The following method can be used to create first branch and work on it:

- Create branch from the master through server (Stash, Bitbucket or Github). Open the already created "hello-git" repository in the stash.
- Select "Create Branch" from the left menu.
- Enter the source branch and the new branch name say, "first-branch". Click on Create Branch button.
- Now open shell in your local cloned repository and run
  git fetch
- Now the created branch can be checked out with the following command:
  git checkout first-branch

The method mentioned above is the recommended method. But once you understand the git a little more the following method can be used which is way faster:

- Open shell in the local repository and run the following command to create a testing branch:
  git branch testing
  The new testing branch has been created, but if we run git status, we'll know that we are still on the master branch. To change branch to testing, run the following command:
  git checkout testing
  The short-hand for both of the above commands is the following:
  git checkout -b testing
- At this point, this branch can be sent to the remote from the local repository:
  git push --set-upstream origin testing:testing
  This will push the testing branch to the remote and starts tracking changes if they happen in the remote branch. It means the local and remote testing branches are being tracked with respect to each other. If someone else changes the remote testing branch and we do git fetch, the git will tell us thorough git status that the remote testing branch has more commits than the local testing branch.
  It should also be noted in the last command that we have put branches names as A:A as the local branch name is A and remote branch name is also A. If we want to push this testing branch to the remote with name testing-2, we can run the following:
  git push --set-upstream origin testing:testing-2
  But different names should be avoided as they create confusion. Remember to use them once you are a regular `git` user.

Branches creation practice

A developer can create as many branches as he wants for his need and push only the needed branches to the remote. Suppose we have created 3 branches from the one master branch. Now we do 1 commit on each branch. Our local repository will look something like the following figure: (Don't worry this figure is a little abstract one.) Example commands to make such tree is:

- `git checkout master`
- `git checkout -b branch0`
- `vim file0.c #Create and save new file`
- `git add file0.c`

- `git commit -m "Add file0.c"`

- `git checkout master`
- `git checkout -b branch1`
- `vim file1.c #Create and save new file`
- `git add file1.c`
- `git commit -m "Add file1.c"`
- `git checkout master`
- `git checkout -b branch2`
- `vim file2.c #Create and save new file`
- `git add file2.c`
- `git commit -m "Add file2.c"`

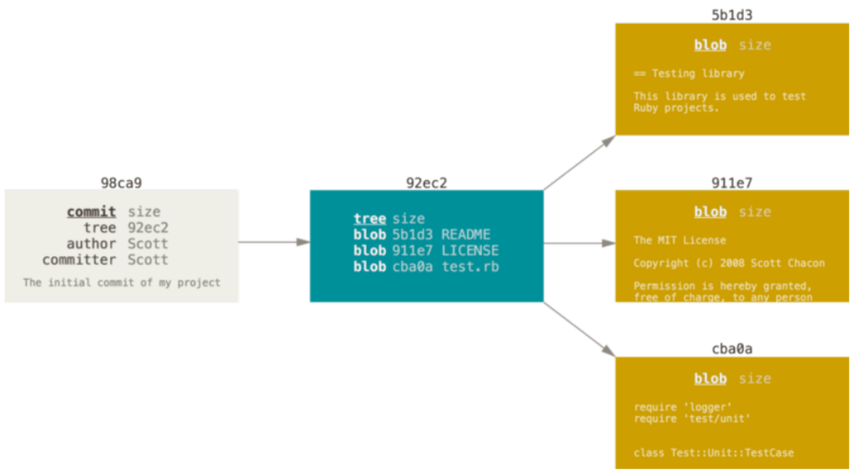


Figure 9. A commit and its tree

Branch names

Following command can be used to find out the local branches:

git branch

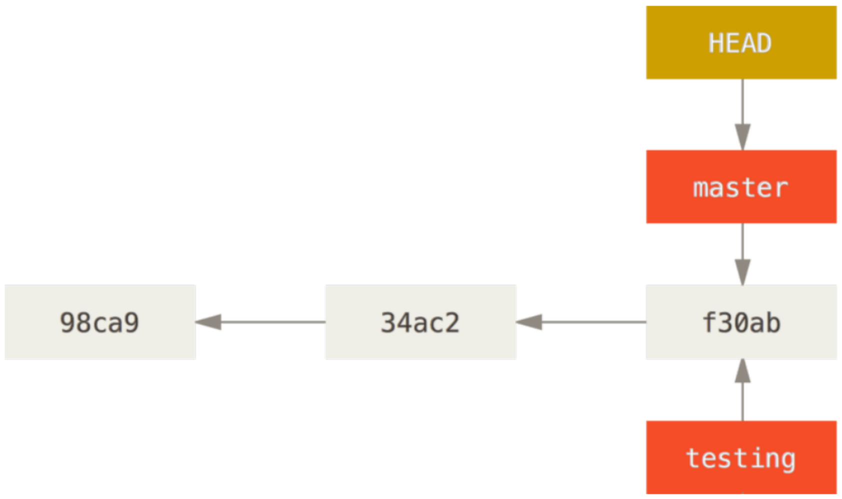
Following command can be used to find out the names of the remote branches:

git branch -v

Commit on a branch explained:

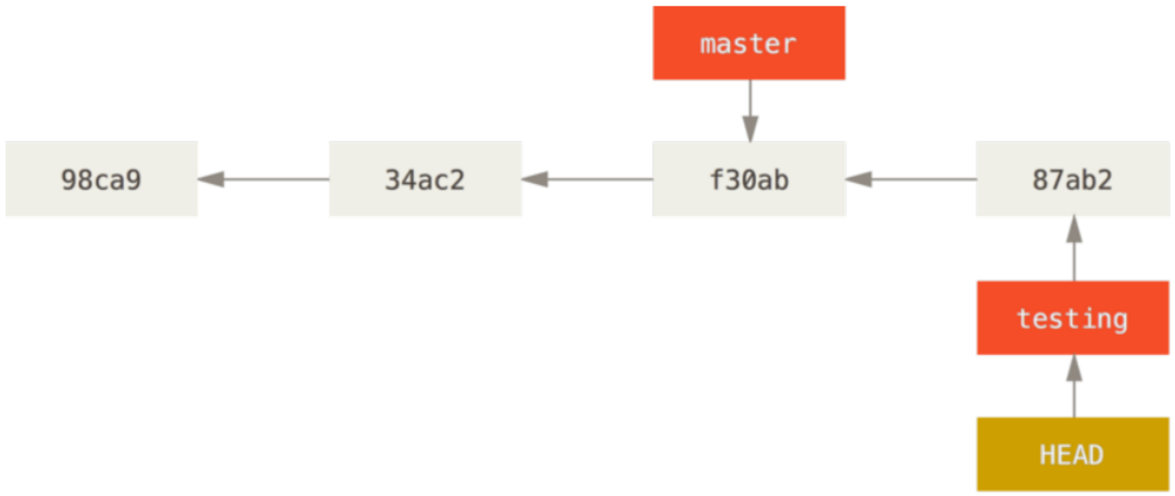
Let's create a local branch from the master and move to it. Remember HEAD is a pointer which shows on which branch we are working on.

git checkout -b testing



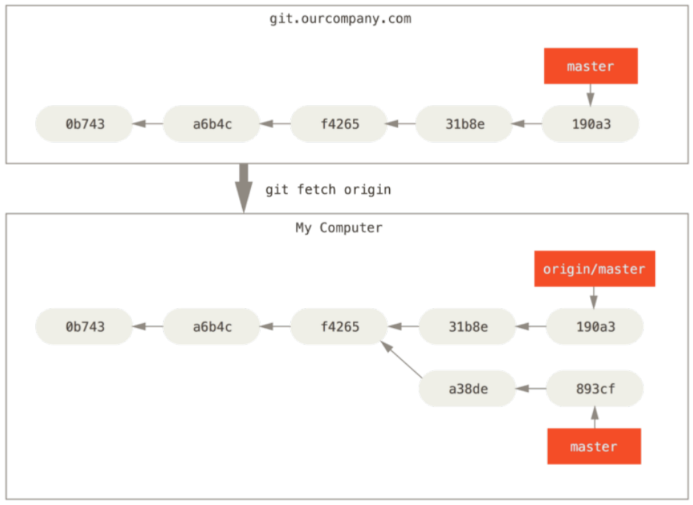
Whenever we do a commit, it is done at the HEAD of the current branch and HEAD also moves to the new commit.

git commit -a -m ?Removed USB bug?



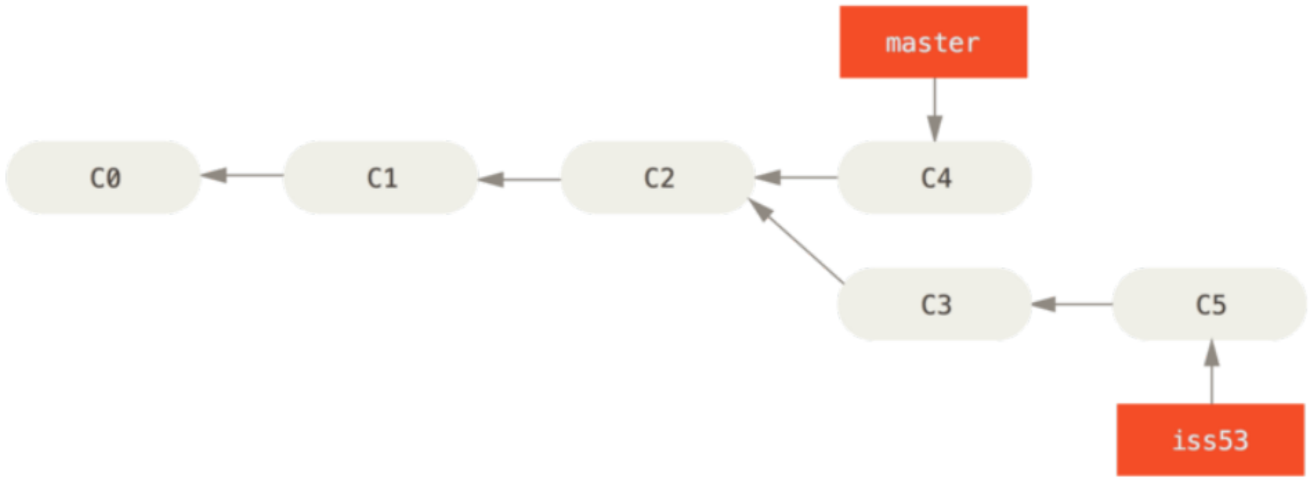
Different branches at remote and local

The remote may have different branches than the branches present in the local repository. Usually people send their local branches to the remote once work is in progress, complete or they want to open a pull request. The following figure shows the local master and master at the remote have diverged in the local repository. (Usually diverging a master from the remote repository is a very weird thing to do as master should always be same as that of the remote. If someone wants to play, he should create local branches as much as he wants.)

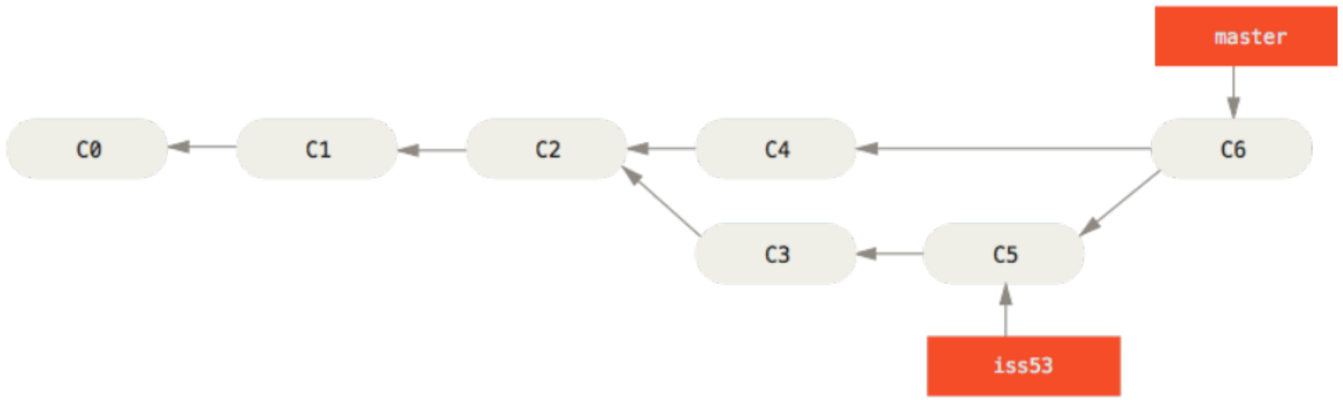


Merging

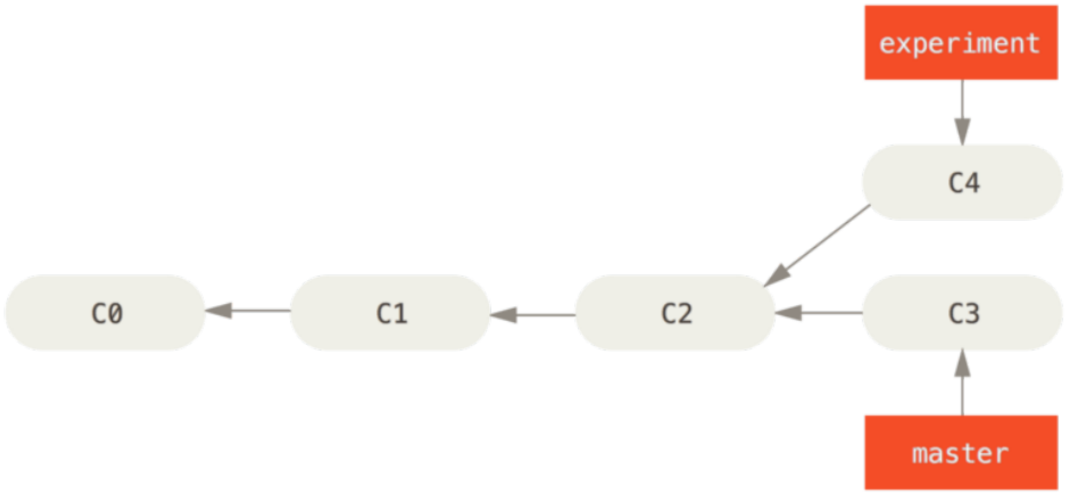
Branches are created with the intention that they will be merged back to the master once work is complete. Let's start looking at merging branches. If master is at the same commit from which we had created our branch, the branch can be merged to the master easily. Let's explore a little different case that how merge happens if master has been updated by someone else while we were working on our branch. Suppose "iss53" is our branch. It was created from master when master was at C2 commit. Now master has C4 commit and "iss53" branch has commits C3 and C5.



Now if we merge "iss53" branch to the master (through server stash etc), another C6 commit will be done which will merge the changes from all C4 and C5 commits as their parents. This kind of merge is also called Non-Fast-Forward Merge. This merge has made the history non-linear. Imagine if 100s of branches merge like this, the history will look horrible.



Let's assume another case where we had created branch "experiment" from the master when it was on C2 commit. Now master has moved to the C3.



If we rebase our experiment branch to the new master, it'll change the history of our branch and it'll start to look as if this branch was created from the master when it was on C3 commit instead of when it was on C2 commit.

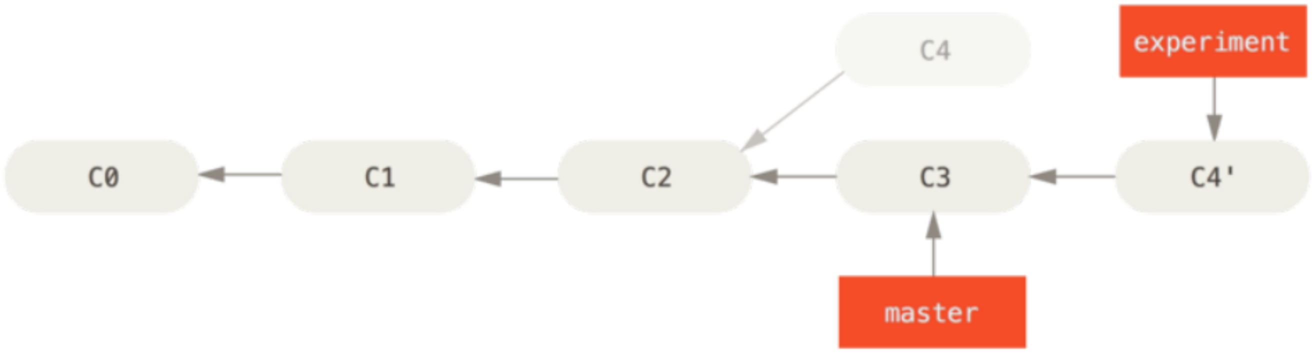


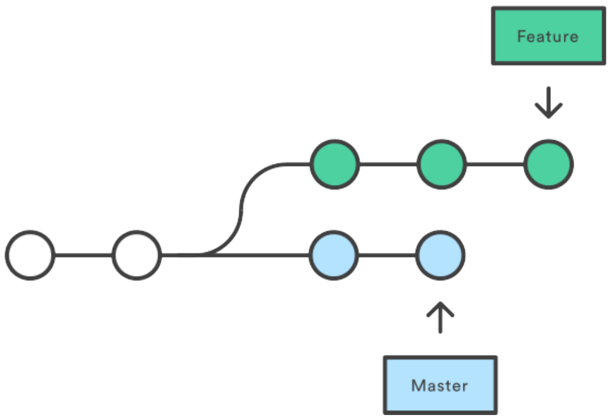
Figure 37. Rebasing the change introduced in `C4` *onto* `C3`

Note that rebasing a branch has changed the hash of the commit from C4 to C4'. (Of course these numbers represent commit hashes.) The history in this way will remain linear.

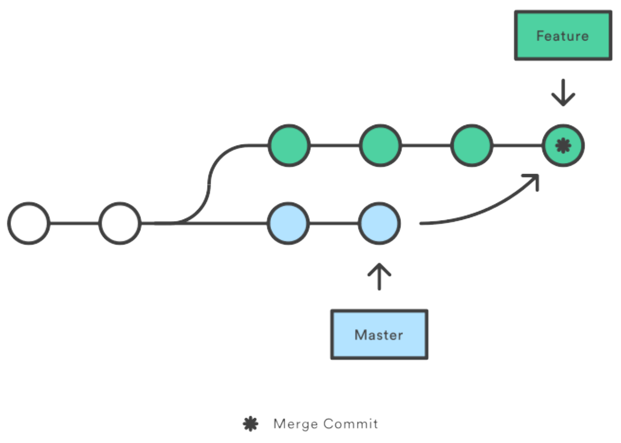
Merging and Rebasing

Let's compare merged and rebased branch merge into the master.

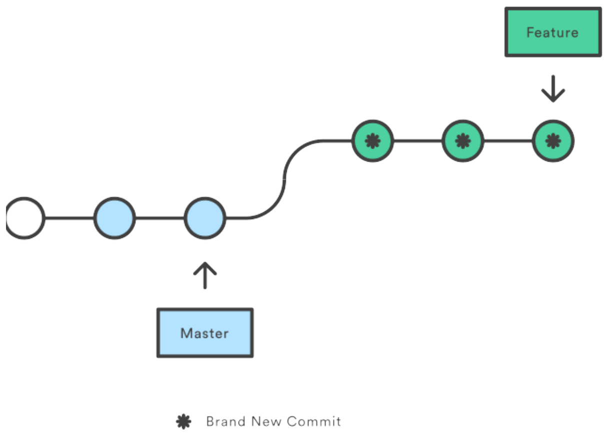
A forked commit history



Merging master into the feature branch

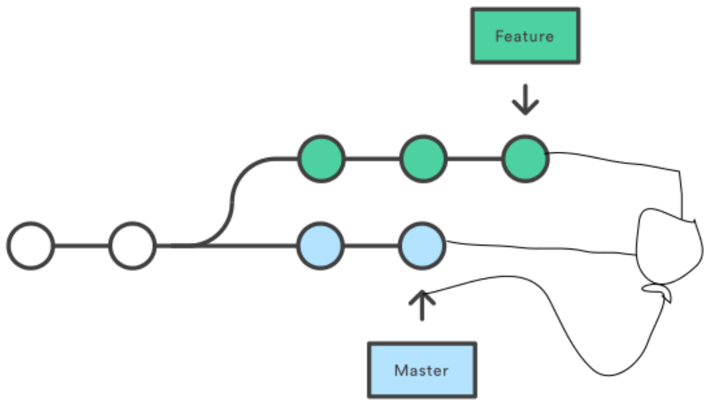


Rebasing the feature branch onto master

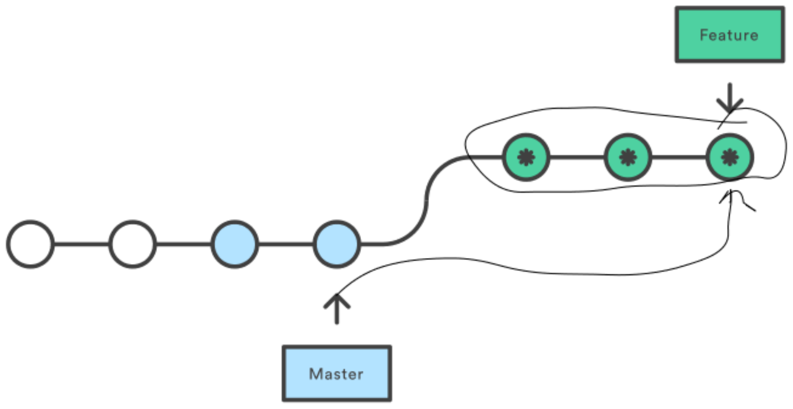


After merging these branches to the master, state will be:

A forked commit history



Rebasing the feature branch onto master



Note: After merging/rebasing sanity testing should be done and then the pull request should be merged.

Practice:

- Open shell in the earlier created repository and checkout master
  git checkout master
- Create first branch and commit a dummy file:
  git checkout -b branch0
  git add dummy
  git commit -m "add dummy"
- Create second branch and commit another dummy file:
  git checkout master
  git checkout -b branch1
  git add dummy1
  git commit -m "add dummy1"
- Usually we don't have permission to play with master. But this is our repository with our rules. Let's merge branch0 into the mater:
  git checkout master
  git merge branch0 #Usually this is done by server when PR is merged
- Now master has moved on. If we merge branch1 as it is into the master, it'll be non-fast-forward merge and history will be non-linear as explained above.

```
git checkout master
git merge branch1
```
Now run git log and git status to understand this merge properly.

- Let's understand the rebasing now. Do all the above steps and leave the previous step. Now lets try merging with rebasing method:
  ```
  git checkout branch1
  git rebase master
  git checkout master
  git merge branch1 #Usually this is done by server when PR is merged
  ```
  Now run git log and git status to understand this kind of merge properly. It is fast-forward merge.

E. Commands to become Regular User

Checking out files:

If there are some temporary changes in a file, those can be discarded with:

```
git checkout -- <file>
```

File from another branch can also be checkout out with:

```
git checkout <branch> -- <file>
```

Diff:

If we have changed a already present file in the repository, we can check the difference of untracked file by:

```
git diff
```

To see the difference of the tracked files: (cached and staged are synonyms.)

```
git diff --staged
git diff --cached
```

File movements:

Git is good with tracking changes across same file names. But it sometimes forgets to compare changes and keep the blame of the file if we don't move it with the following command:

```
git mv file_from file_to
```

Always move already present files in a repo with this command. Also use this command if you want to rename some files.

Deleting Branches

Local branches should be deleted from time to time. Keep in mind to delete a branch while staying on another branch.

```
git branch -d <branch>
```

```
git branch -D <branch> (if branch isn?t merged)
```

Deleting Remote branch

Branches from remote repositories can be removed by executing a command from local shell. This is my favorite feature. Don't delete other's branches though.

```
git push origin --delete <branch>
```

```
git push origin :<branch>
```

Git clean explained:

Sometimes there are useless files in a repository like .cproject or .project files which are created by some software in our repository. This kind of untracked files can be removed using git clean command.

```
git clean -n -d -f
```

Git's internal aliases

Git also allows to create internal aliases. An example can be seen above.

```
git config --global alias.unstage ?reset HEAD --?
```

```
git unstage fileA
```

Commands I use on daily basis - Practice:

1. I create a branch in local repository
   git checkout master
   git checkout -b crash-fix
2. I edit files to remove the bug (of course by debugging, spending time to find out this bug and after regorous testing).
3. I check difference of the changed files whenever I want to know how many and what I've changed in the source code in the whole repository:
   git diff
4. I stage the changed files
   git add abc/buffer.c
5. I check if I've staged the correct change-set
   git diff --cached
6. I do the commit
   git commit -m "Fix the crash in kernel"
7. AT this point, I realize that there are many cosmetic errors in my last commit. SO I fix the cosmetic changes (Edit files again)
8. I check difference of the changed files to see if I've fixed cosmetic changes correctly.
   git diff
9. I stage the changed files
   git add abc/buffer.c
10. I check if staged changes are correct ones:
    git diff --cached
11. I do the commit
    git commit -m "Cosmetic changes"
12. After doing two commits, I usually find that I've missed a dot or misspelled a word in the comment. So I do 2 more commits in the same way.
13. Now there are 4 commits. I can verify it by:
    git log
14. I'll push this branch to the remote now. So that I can open a PR:
    git push
    But git push will fail as remote doesn't have this branch. So I'll create a branch in the remote and push my changes with the following command:
    git push --set-upstream origin crash-fix:crash-fix
15. After opening PR, I'll feel embarrassed that I'd done a commit to just add a dot. How foolish will people think I am!
    (I should learn to amend a commit or squash commits to avoid embarrassment. So keep on following this tutorial to learn those.)

Global and Local Configurations

Git can have local or global configurations. Local repository has configurations which apply to just that local repository. Usually global configurations are set once and then those configurations get applied to all the repositories.

Name and Email setup

```
git config --global user.name ?johnwick?
git config --global user.email jhonwick@me.c
```

Core editor to write commit messages etc

The default editor is vim sometimes which may create problems for beginners. So use nano or notepad++ etc.

```
git config --global core.editor <Path to editor>
```

Credentials helper

The wincred is best for Windows and cache is best for Linux.

```
git config --global credential.helper <cache/wincred>
git config --global credential.helper 'cache --timeout=300'
```

Typo command helper

If you write git stats, it is a typo command. You can set the auto correct config and set the timeout. So after setting this config, git stats will be corrected and executed automatically as git status and thus saving our precious time.

```
git config --global help.autocorrect 50??????? # 50 = 5 seconds timeout
```

Help in resolving conflicts

This config may help if there are some repetitive conflicts.

```
git config --global rerere.enabled true
```

Git LFS

Git isn't good with storing the large binary objects like PDFs or binaries of the compiler etc in a source code repository. So git lfs is used for this kind of files. Usually administrators have already set that which files will be tracked by git lfs. So most of times, developers deal with these objects just like usual files. Only the following config is needed sometimes for smooth lfs usage.

```
git config --global lfs.contenttype 0
```

SSL Verification

SSL certificate should be added or if certificate cannot be added the following option can be helpful sometimes:

git config --global http.sslVerify false

F. Commands to become a Power User

Ancestry reference and Git show

"^" or "~" symbols are used for ancestry references. Git show command is used to view the commit as a whole which includes the log and the diff of that commit. Let's view the commit on the HEAD:

```
git show HEAD
```

Use the following command to view the parent of the current commit:

```
git show HEAD^
```

Use the following command to view the 4th commit in the hierarchy.

```
git show HEAD~4
```

It is really important to understand the use of these symbols as they are highly used in git.

Git ranges - Double Dot Notation

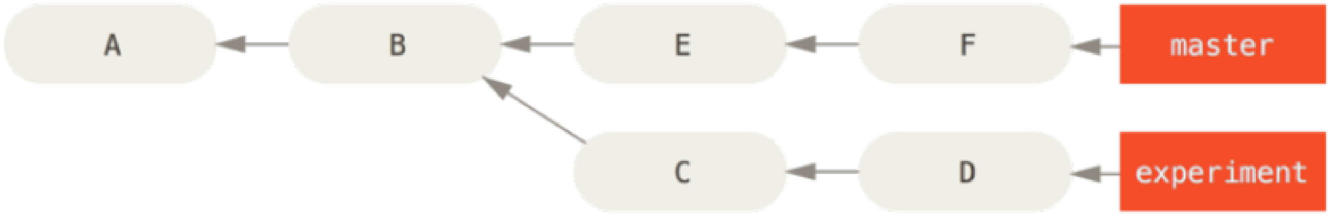
What is in experiment branch that isn't present in the master branch?

```
git log master..experiment
D
C
```

```
git log experiment..master
F
E
```

In the same way diff can also be seen:

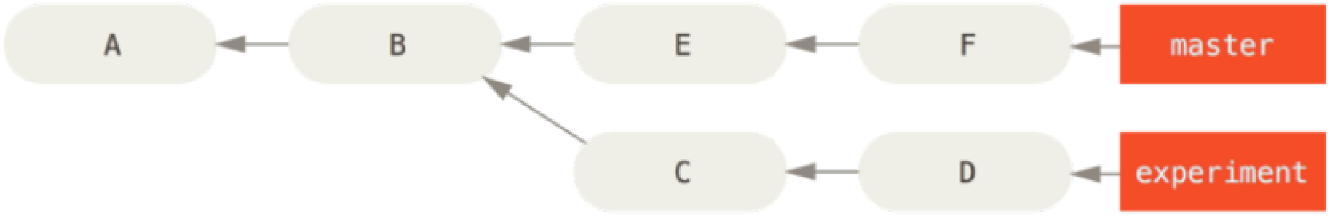
git diff master..experiment



Git ranges - Triple Dot Notation

Which commits are different in both branches?

```
git log master...experiment
F
E
D
C
```



Rewriting History - Git commit amend

Commit amend is used to amend the last commit?s contents or just the commit message.

- Assume we have done a commit, but we had forgot to add one file. So we'll stage that remaining file and shall run:
  git commit --amend
  This'll add this file to the commit and will also open the editor to edit the commit message also.

Note: Practice this amending process as this will increase your productivity a lot later on.

Rewriting History - Interactive rebasing

Interactive rebasing is used to:

- Squash
- Drop
- Edit
- Reword
- Change order of the commits

Assume we want to squash the last 3 commits into 1 commit. Run:

git rebase -i HEAD~3

An editor will be opened. Change the "pick" with "squash" on the left most column against the last 2 commits and close the editor. These 2 commits will be squashed into the first commit. Then a editor will open again asking for the updated commit message. Edit it and close it.

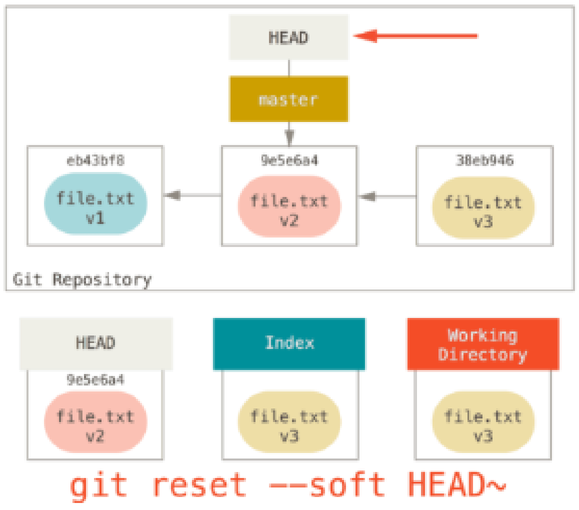
Note: These two operations (git commit --amend and git rebase -i ...) change history. Use them carefully. Beginners should learn them carefully. So after performing them please push them forcefully:

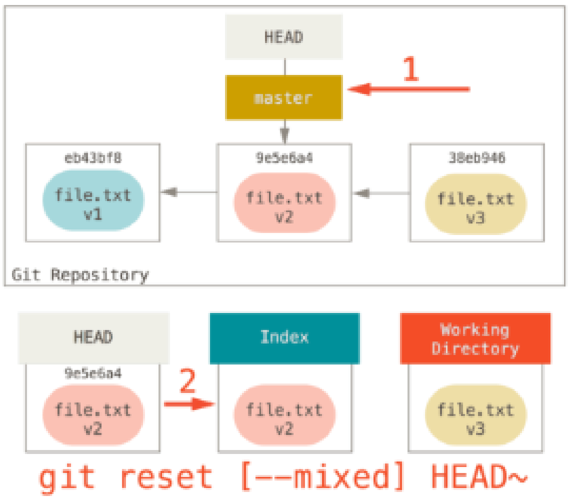
git push -f

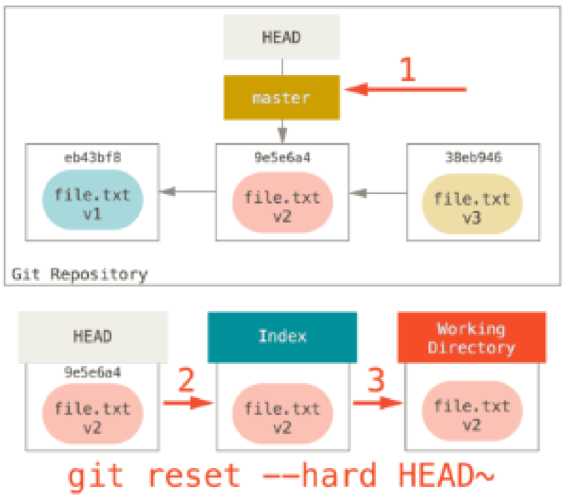
Git reset explained

Git reset is of three types:

1. git reset --soft
   This'll move the pointer of the current branch to the specified commit one, and "Working directory" and "Index" remain same.
2. git reset --mixed
   This'll move the pointer of the current branch to the specified commit one, and "Working directory" remain same.
3. git reset --hard
   This'll move the pointer of the current branch to the specified commit one. This is used to move head and kind of discard the commit.







Conflicts removal

If two persons have changed the same line of same file, git may not be able to merge the two files by itself. So git puts both changes in that file and asks the developer to keep one or merge both of changes manually. This is called conflicts resolution. Suppose we rebase or merge a branch:

```
git <merge/rebase> master
```

Git will tell us that the conflicts have arisen. It cannot complete the merge/rebase. We can find out about the conflicted files by running

```
git status
```

The conflicted file contains something like the following:

```
<<<<<<< HEAD
INT I = 10;
=======
INT J;
>>>>>>> new_branch_to_merge_later
```

Keep one of the portion or merge them depending upon the source code. Remove all <<<<, ==== and >>>> symbols.

Stage the files and continue the merge/rebase:

```
git add <file which had conflicts>
```

```
git <merge/rebase> --continue
```

If something goes wrong or you don't want to merge/rebase at any stage, abort it:

```
git <merge/rebase> --abort
```

Interactive Staging

Interactive staging is very powerful feature through command line. But most of the times people use some GUI software to do interactive staging.

```
git add -i
```

Then follow the steps :D.

Stashing

Local changes can be stashed which we don't want to commit or if we want to put them somewhere away for a while. Stashing is done as if we have a stack on which changes will be stored. So last thing to go into stashing will come out first when we?ll pop it. Learning git stash and git pop should be enough and they help a lot later on.

```
git stash
```

```
git stash pop
```

```
git stash list
```

Search inside git

```
git grep sabre_lite
```

```
git log --grep=<pattern>
```

Reverse Commit

Git revert does the reverse commit of the mentioned commit. This method of reverting a commit is discouraged. This command is only used if one want to revert something from the master.

```
git revert <commit-hash>
```

Git cherry-pick

Git cherry pick is used to bring a commit from another branch into the current branch?s HEAD.

```
git cherry-pick <commit-hash>
```

G. Advanced command set (DIY)

This section hasn't been written in detail as these concepts can be consolidated later on. The purpose of this section is to give you a clue that these features exist in git and you can use them.

The life saver - Git reflog

What to do if destructive operation gets wrong? The discarded commits etc can be found in the reflog of the git. It can help sometimes.

```
git reflog
```

Garbage collection

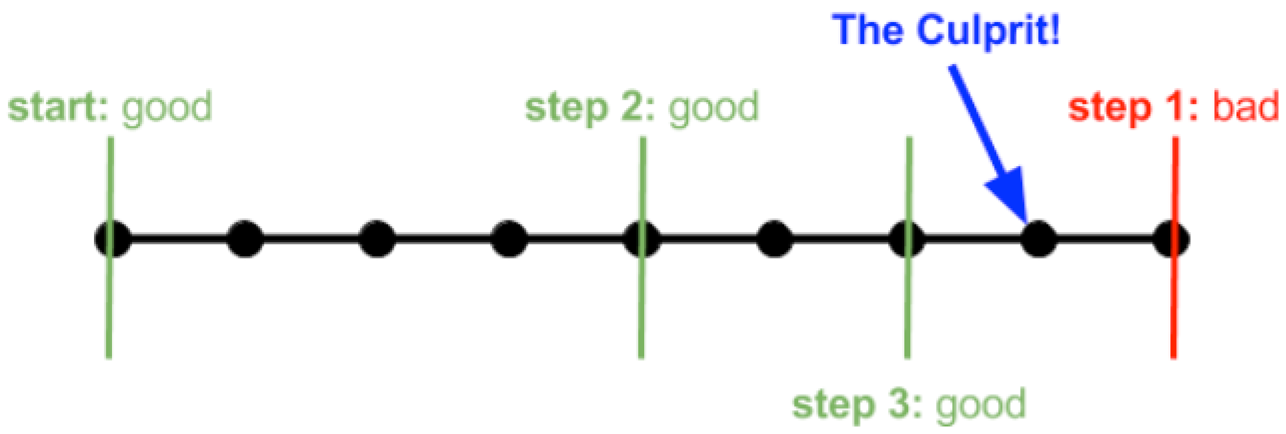
Git gc is used to do garbage collection. It also increases the speed of an older repository.

```
git gc
```

Finding Malicious Commit

Sometimes after merging certain number of commits, the system starts to crash and we don't know which commit has broken the system. So git bisect comes for rescue.

```
git bisect start
git bisect bad <hash>
git bisect good <hash>
git bisect skip
git bisect reset
```



Git Sub-Modules

It often happens that while working on one project, you need to use another project from within it. Perhaps it?s a library that a third party developed or that you?re developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other. Git submodules are used in this case.

git submodule add https://mentor.com/tf.git
git push origin master
git clone https://mentor.com/nuc4.git
git submodule init

Git Hooks

Hooks are scripts which can be enabled to run when we run some command of git. Hooks are used by power users. They must be explored later on. Some most used hooks are:

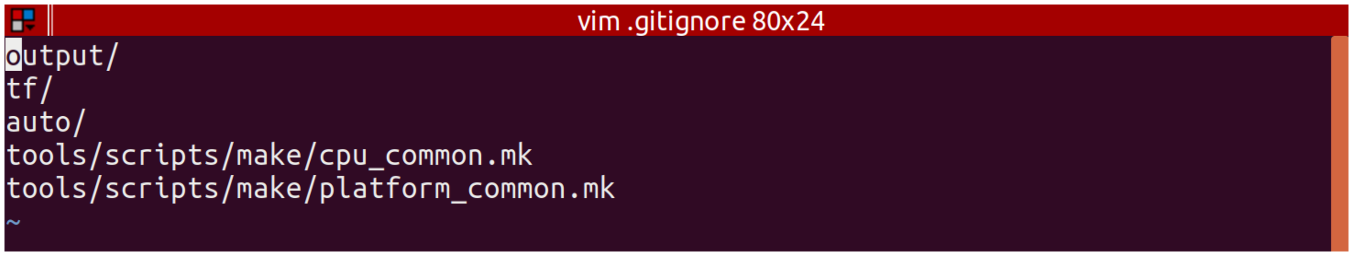
- pre-commit
- pre-receive
- post-commit
- post-receive

Hooks should be added to .git/hooks/ if you want to use them.

H. Ignore Useless Files

There are plenty of useless files in the perspective that we don't want to save them. For example, we don't want to keep the build artifacts in our repository. The patterns of these files names can be ignored to exclude from the repository entirely. These files/folders get changed or not, git will never be bother by them.

Usually .gitignore file is present in the root of the repositories. The blob patterns of the files which we want to ignore are specified in this file.



vim .gitignore 80x24

```
output/
tf/
auto/
tools/scripts/make/cpu_common.mk
tools/scripts/make/platform_common.mk
~
```

I. References

Pro Git book, written by Scott Chacon and Ben Straub, https://git-scm.com/book/en/v2

Getting Git Right blogs by AtLassian, https://www.atlassian.com/git