CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 3

Tuesday, September 14th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- Shell customization
- Input/Output (I/O) and redirection
- Job/process management
- Environment variables
- Shell scripting (today)

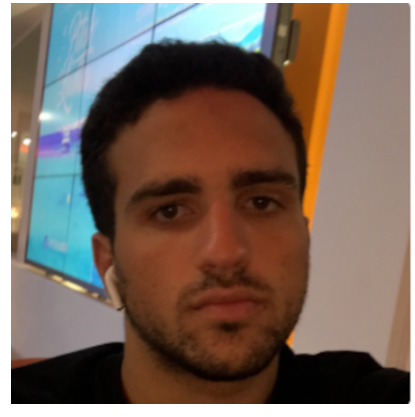
OUTLINE

- Introduction to version control systems
- Centralized and distributed approaches
- Bare essentials of `git`
- Interactive `git rebase` demo

Content and some figures are based on the free Pro Git book written by Scott Chacon and Ben Straub.

NEW TEACHING FELLOWS

- Sergey Litvinov
- lisergey@seas.harvard.edu



- David Assaraf
- davidassaraf@g.harvard.edu

PP-SECTIONS IN-PERSON

- Pair-programming sections are now mostly ***in-person***. This is not only more efficient for the supervising TF but also more efficient for your team progress.
- We continue to offer two sections via zoom mainly for students in self-quarantine.
- We have also added *two more* office hours. See the main website https://harvard-iacs.github.io/2021-CS107/

Pair-Programming Sections

Time given corresponds to *starting time*. Duration of a section is 75 minutes.

> Attendance of one pair-programming session per week is **mandatory**.
>
> The sections offered on zoom are to be prioritized by students who need to self-quarantine or are unable to attend in-person otherwise.
>
> - **Friday** (start of pair-programming cycle with *first* section of the day)
>   - 11:00am - 12:15pm: SEC 6.412 (Connor)
>   - 4:00pm - 5:15pm: SEC LL2.224 (Sehaj)
> - **Monday**
>   - 5:00pm - 6:15pm: Johnathan's Zoom
> - **Wednesday**
>   - 5:15pm - 6:30pm: SEC 1.413 (Yang)
> - **Thursday** (end of pair-programming cycle with *last* section of the day)
>   - 10:30am - 11:45am: SEC 1.316 (Erick)
>   - 5:00pm - 6:15pm: David's Zoom (starting from: 09/23/2021)

INTRODUCTION TO VERSION CONTROL SYSTEMS

INTRODUCTION TO VERSION CONTROL SYSTEMS

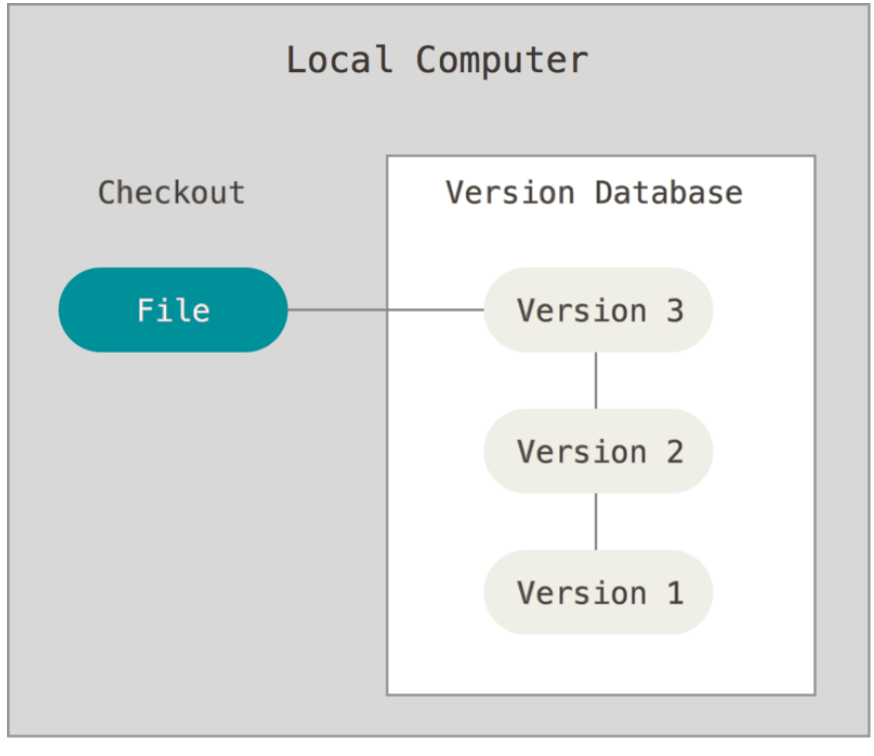
WHAT IS VERSION CONTROL?

Version control is a system (abbreviated VCS) that records changes to a file or set of files over time so that you can recall specific versions later.

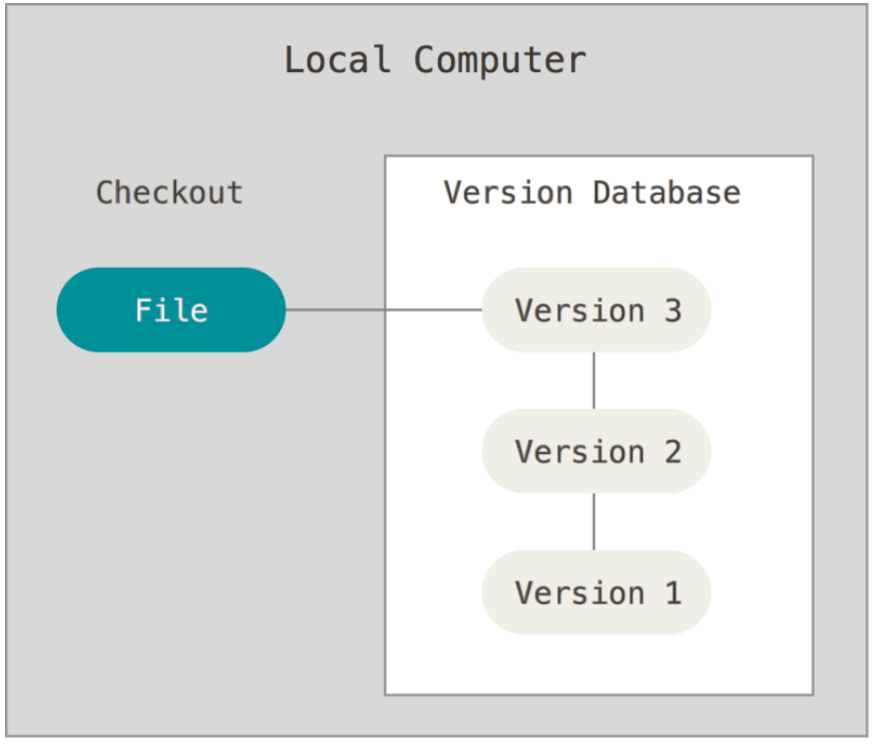
- VCS is absolutely *essential* if you plan to do serious software development *(academic and company based software development)*
- If you work on a software project with more than one person*, not having a VCS will just **not** work! (*note that it should actually read *more than zero*)

LOCAL VERSION CONTROL

If you would only need version control on your local machine (e.g. laptop), you would probably make backup copies of your files into some directory you dedicate for VCS.



LOCAL VERSION CONTROL



Discuss with your neighbors:

If you were to implement such a tool with a shell script, which of the available basic Linux commands would you use for this task?

LOCAL VERSION CONTROL

Discuss with your neighbors:

If you were to implement such a tool with a shell script, which of the available basic Linux commands would you use for this task?

- One command that would be useful for this task is `diff`. You can track the *difference* between two files and store this data as *patches*. You can then reconstruct the state of a file at an earlier time by *recursively* applying patches in reverse.
- You could also use the `rsync` tool together with `ln` to form hard-links. This strategy is similar to *differential backups* but would likely require more disk space than storing simple diffs only.

LOCAL VERSION CONTROL

- Doing all of this through a shell script is, of course, *error prone and associated with quite some overhead* to get the script to a state with production quality.
- Additional complexity will be added when you must account for *multiple users* which are possibly not working on the same machine. (Each user has a local VCS database, how do you sync it?)
- There are many tools that can do the job much more efficient, with a minimal extra overhead for you to maintain your code/files.

EXAMPLE FOR VCS IN ACTION

Many people maintain a directory with all sorts of configuration files for tools, such as `bash`, `zsh`, `vim`, `git` and others. Here is a snapshot of my `.rc` directory, which is under VCS as seen in the prompt:

```
+[fabs:.rc] master ± ls -a
.                     .conkyrc              .inputrc              .style.yapf
..                    .conkyrc-colors       install_helpers.sh    suckless
.abcde.conf           .conkyrc-colorsGIT    install_minimal.sh    .taskrc
base16                .ctags                install_rc.sh         .tmux.conf
base16-manager        .dircolors            jabref.xml            .tmux.conf_osx
base16-README.md      .dircolorsrc          .jnewsrc              uninstall_minimal.sh
base16-shell          .emacs                .liquidpromptrc       .unison
.bash_aliases         .ethsync              .liquidprompt.theme   .vifm
bash_completion.d     .fzf_tmux_helpers     mail                  .vim
.bash_profile.brutus  .gdbinit              modulefiles           .vimpagerrc
.bashrc               .git                  .mutt                 virtual_machine
.bashrc_macOSX        git                   pacman                .xbindkeysrc_lenovoT14
.bashrc_wrapper       .gitconfig            paraview.pwin         .xinitrc
.bashrc.xubuntu       .gitignore            .profile              .Xmodmap
bin                   .gitmodules           qutebrowser           .Xmodmap_linuxHHKB
borg                  .git-prompt-colors.sh README                .Xresources
.calcurse             .git_template         .ripgreprc            .Xresources.d
.clang-format         gnupg                 .slrnrc
config                .gnus.el              ssh
+[fabs:.rc] master ±
```

EXAMPLE FOR VCS IN ACTION

Why do I care about having these files under VCS?

- I have **accounts on multiple machines**. E.g. my laptop, remote clusters, RaspberryPi, etc.
- I spend more time on one machine than others, but when I am working that other machine, I want my **configuration to be identical**.
- My config files **change often**. I learn something new that I think is useful, I will integrate it in my workflow.
- Because I am using VCS, I can make such changes **irrespective** of the machine I am currently working on. Any changes propagate to all the other machines by *pushing the changes to a remote hub*.
- A year later I can browse through my VCS history and reflect the old times while thinking to myself *"I can't believe I did such a complicated thing back then..."*

EXAMPLE FOR VCS IN ACTION

You can still be flexible, even with VCS:

- My main `.bashrc` file is under VCS in the `$HOME/.rc` directory
- On each machine you create the `~/.bashrc` file which *wraps* your main `.bashrc` file that is under VCS:

```
 1 set -o vi
 2
 3 # source your main .bashrc configuration (under VCS)
 4 source ~/.rc/.bashrc
 5
 6 # local machine specific environment variables, possibly override vars if ne
 7 export EDITOR=vim
 8 export GIT_EDITOR=vim
 9 export VISUAL=vim
10 export BROWSER=qutebrowser
11 export DEFAULT_PDF=zathura
12 export PDFVIEWER=zathura
13 export LOCAL=$HOME/local
14 ...
```

WHY VCS IS A *REQUIREMENT*

- Every change you commit is tracked (author, time, changelog)
- You can easily revert changes
- The VCS allows you to *bisect* your commits in case you need to isolate a bug (e.g. `git-bisect`)
- You know exactly *when*, *where and who* introduced changes. This is crucial in large projects with many developers.
- Think carefully about changes you commit in large projects, your name tag is on it and people will know. ***This must not scare you, but raise awareness to become a better programmer!***

EXAMPLES OF VERSION CONTROL SYSTEMS

Concurrent Version Systems (CVS)	http://cvs.nongnu.org/
Subversion (SVN)	https://subversion.apache.org/
Helix Core (proprietary)	https://www.perforce.com/products/helix-alm
Bazaar	https://bazaar.canonical.com/en/
Darcs	http://darcs.net/
Git	https://git-scm.com/
Mercurial	https://www.mercurial-scm.org/

Centralized VCS

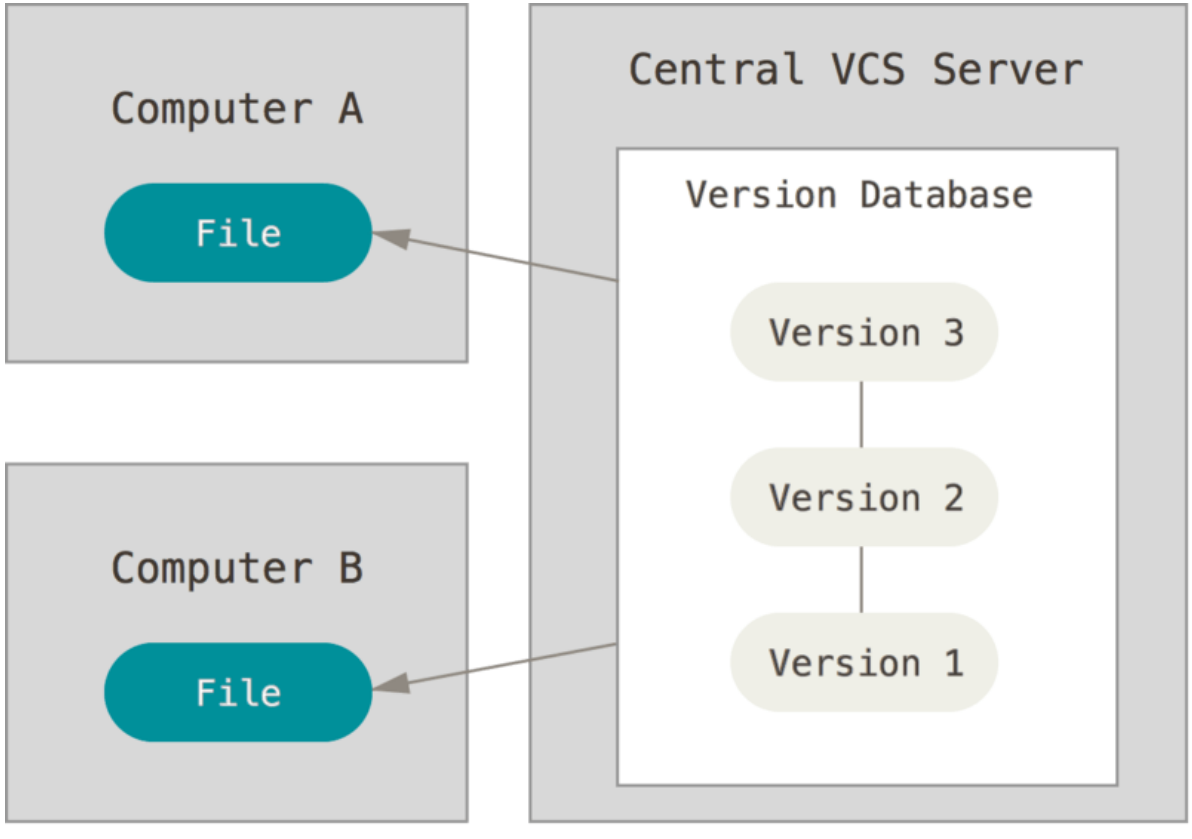
Distributed VCS

Google Drive and Dropbox are **no** examples of VCS

CENTRALIZED AND DISTRIBUTED MODELS FOR VCS

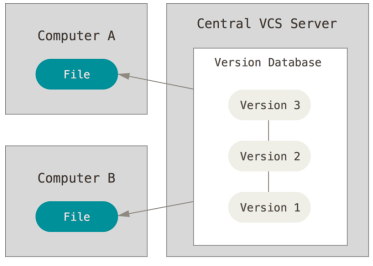
The two main VCS modeling approaches for *collaboration*

CENTRALIZED MODEL (CLIENT-SERVER)



- Instead of hosting the VCS database *locally*, it is hosted on a central server
- Administrators have fine grained control over access rights and policies
- Every one knows to certain degree what everyone else does

CENTRALIZED MODEL (CLIENT-SERVER)



Workflow in a centralized VCS (CVCS):

1. A developer checks out a file to do work on.

   What should be the policy for other developers who want to checkout the same file?

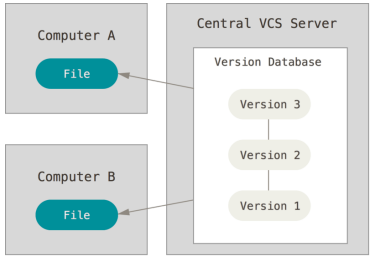
   i. *File locking*: local read-only, write access through locking.
   ii. *Version merging*: local read- and write-access, conflicts resolved through merging algorithms.
2. *Commit* the changes and check-in the file on the central repository:
   i. *File locking*: the check-in process is trivial. The updated file is simply added.

      What is the main drawback?

   ii. *Version merging*: if multiple developers modified the same file, individual changes must be merged. Can be an expensive operation for large changes.

This slide shows examples of different CVCS

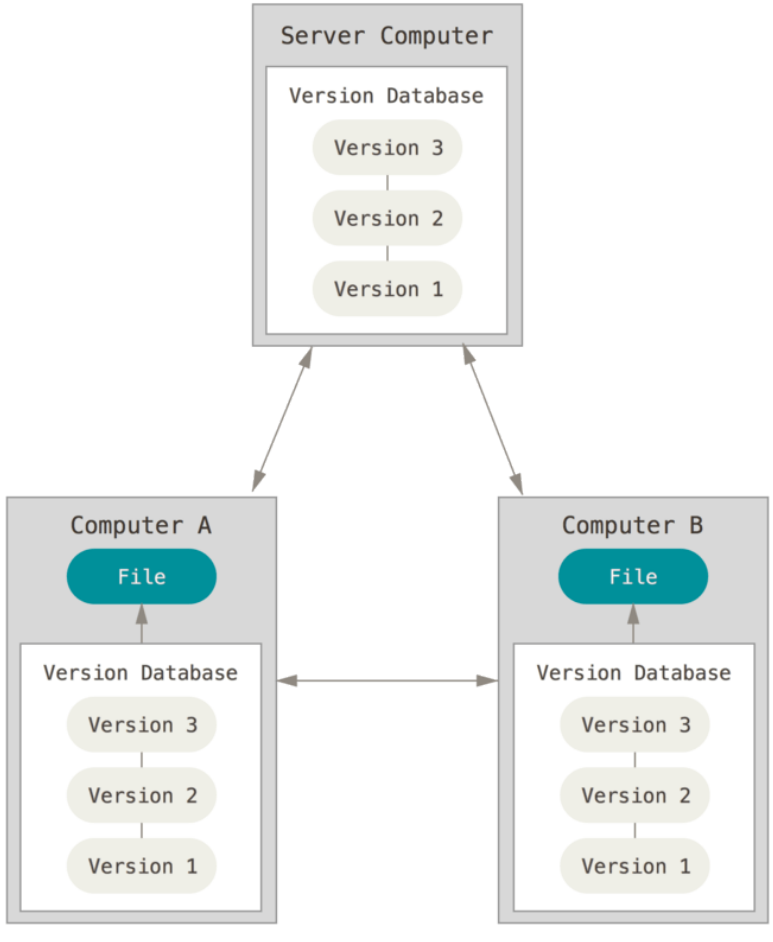
CENTRALIZED MODEL (CLIENT-SERVER)



There are problems with CVCS however:

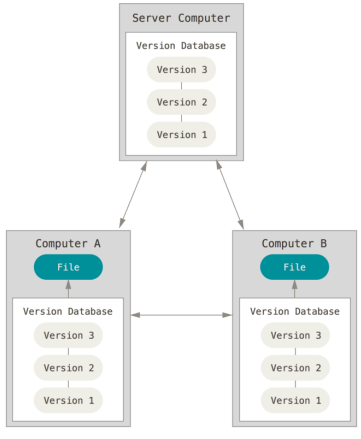
- **Single server instance.** If the server dies, you better have a backup!
- During server downtime (or you are offline), your whole team enjoys a free break, with coffee on the house.
- *Branching* for feature testing and implementation is not trivial in CVCS.
- Resolving merge conflicts can be an expensive problem.

DISTRIBUTED MODEL



- Clients do not only checkout the latest snapshot, but *mirror the full* repository
- If a server instance dies, everybody has a backup locally
- It allows you to work completely offline
- Having multiple server instances is trivial and allows for *hierarchical* collaboration

DISTRIBUTED MODEL



Workflow in a distributed VCS (DVCS):

1. A developer *clones* a repository if it does not exist locally yet.
2. Modifications are committed on the *local* repository. There are two ways how these commits can be distributed to collaborators:
   i. Sending *patches* to collaborators via email. Receivers then patch their local repositories *(you would not need a remote server computer in this case)*.
   ii. Hosting the repository remotely where commits can be *pushed* to or *pulled* from *(as in the figure on the left)*.

What do you think is a potential difficulty in the distributed workflow?

This slide shows examples of different DVCS

BARE ESSENTIALS OF `git`

WHY `git` EXACTLY?

We have seen that there are many alternatives. The existence of `git` emerges from the Linux kernel development:

- The Linux kernel is a very ***large*** project
- There are ***many*** contributors developing ***in parallel***
- On many ***feature branches***

Linus Torvalds created `git`. The home of `git` is https://git.kernel.org/pub/scm/git/git.git/

WHY `git` EXACTLY?

These requirements are summarized in the following key characteristics of `git`:

- Speed
- Simple design
- Strong support for non-linear development *(thousands of parallel branches)*
- Fully distributed
- Ability to handle large projects efficiently in terms of speed and data size

Side note: before `git`, the Linux kernel used BitKeeper which changed licensing policies to commercial focus in 2005 (the year `git` was born). Now the Linux kernel uses `git` and BitKeeper is open-source...

`git` REFERENCES

Recommended `git` references:

- Excellent Atlassian tutorial: https://www.atlassian.com/git/tutorials
- `git` get started on GitHub: https://docs.github.com/en/get-started/quickstart/set-up-git
- StackOverflow beginners guide: https://stackoverflow.com/questions/315911/git-for-beginners-the-definitive-practical-guide

Selection of projects and companies that use `git`:

THE BARE ESSENTIALS OF `git`

BEFORE WE START:

The first steps you should take when working with `git` is to ensure that your contact information is setup correctly. Every commit contains the author information with an email address. All your `git` customization is done in the file `~/.gitconfig`. Verify that the file exists and you have at least the following lines:

```
[user]
    name = FirstName LastName
    email = you@domain.com
```

THE BARE ESSENTIALS OF `git`

BEFORE WE START:

The first steps you should take when working with `git` is to ensure that your contact information is setup correctly. Every commit contains the author information with an email address. All your `git` customization is done in the file `~/.gitconfig`. Verify that the file exists and you have at least the following lines:

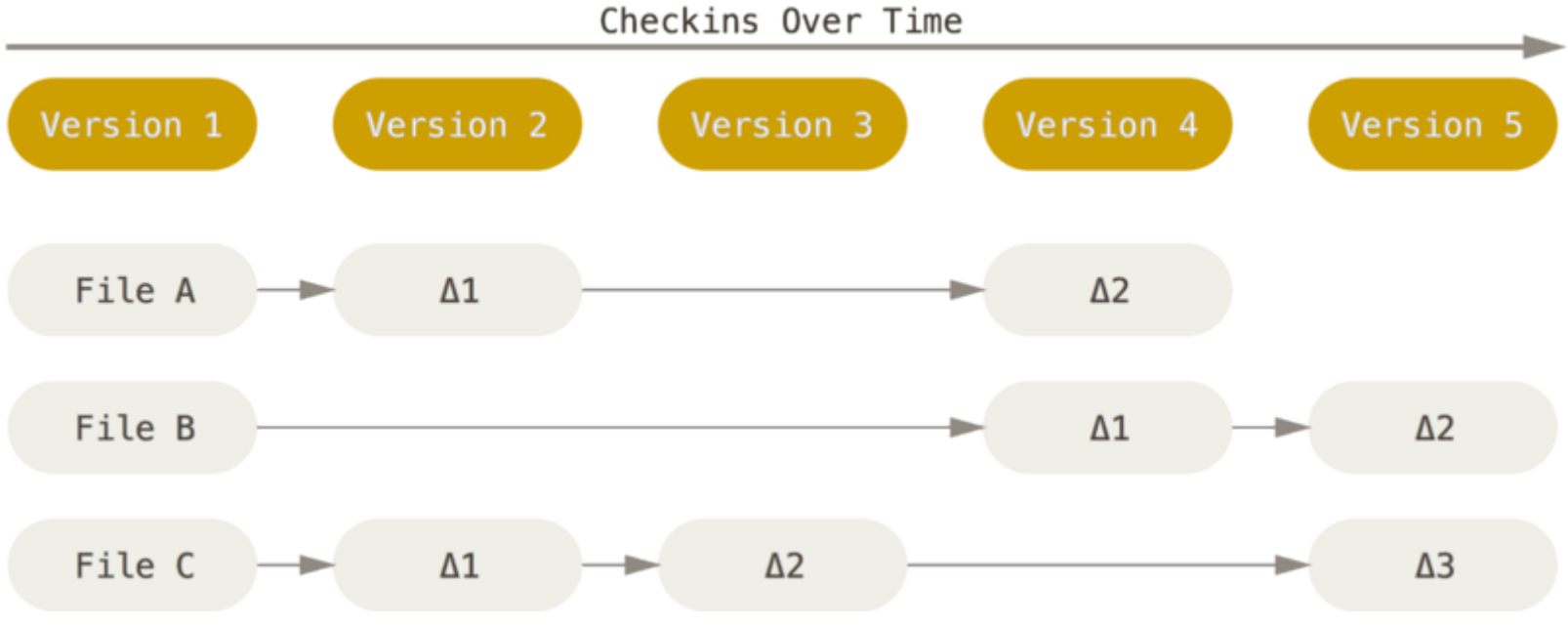
```
[user]
    name = FirstName LastName
    email = you@domain.com
[core]
    editor = vim  ; and possibly this
```

THE BARE ESSENTIALS OF `git`

The major difference between `git` and any other VCS is the way `git` thinks about its data

- Most other VCS store the data as a ***list of file-based changes***
- We refer to this as ***delta-based*** version control
- Think of "delta" as a *difference*, often denoted by the Greek Delta Δ
- ***Recall the earlier slide (local VCS):*** using the `diff` command to implement a VCS system is a Δ-based approach.

THE BARE ESSENTIALS OF `git`



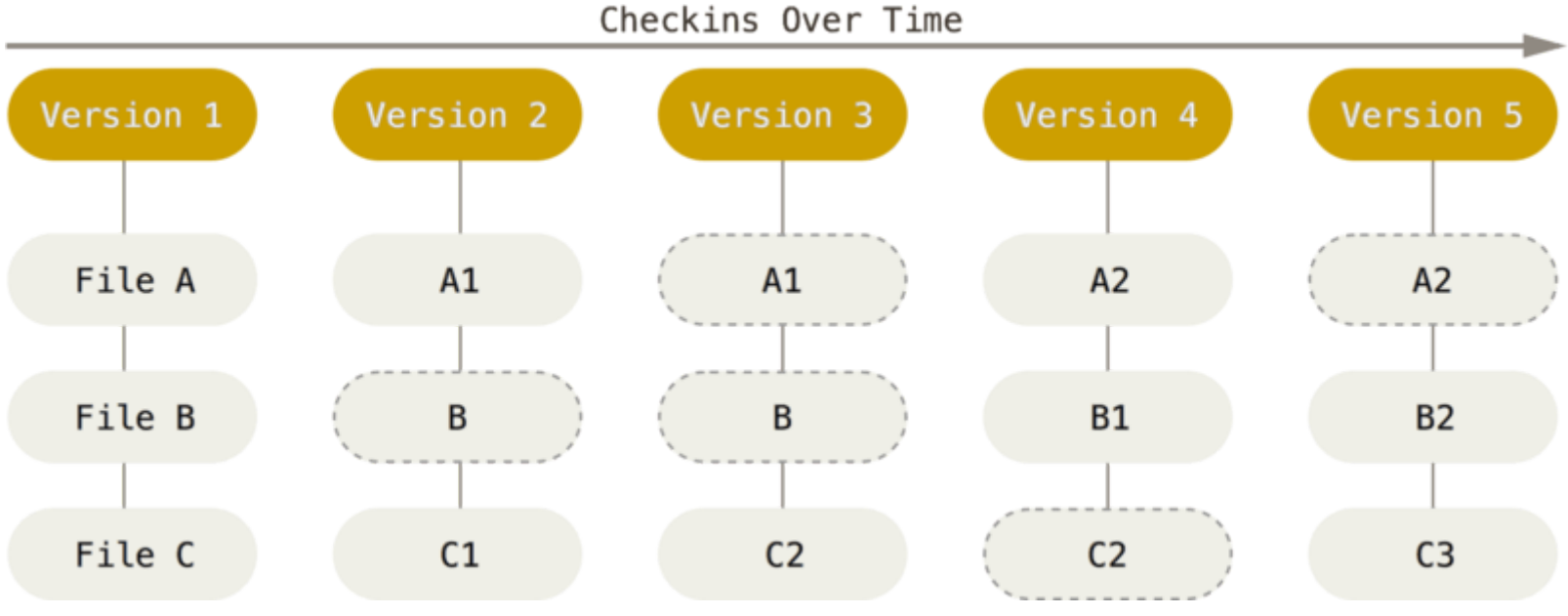
Storing data as a series of differences relative to a base version (delta-based VCS)

THE BARE ESSENTIALS OF `git`

`git` thinks about its data more like a file system and it stores the *state* of the full file system for a particular instant in time.

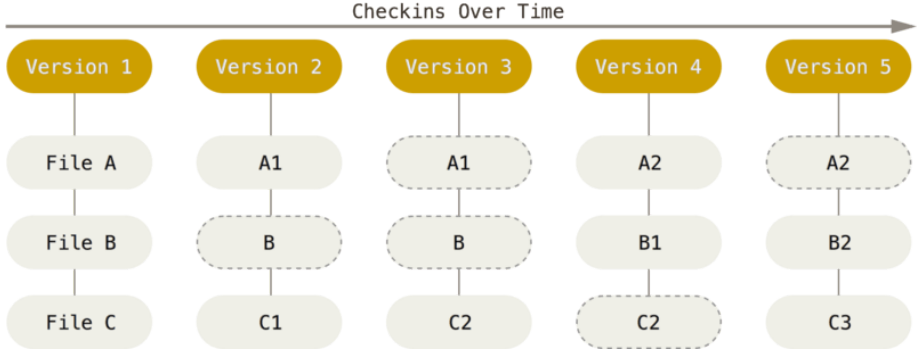
- Such a frozen state is called a ***snapshot***
- Think of it as if `git` is taking a picture of all your files whenever you add a commit
- If a file has not changed `git` will not store it again but simply add a ***reference*** to the already stored file
- ***Recall the earlier slide (local VCS):*** using the `rsync` command (with hard-links) to implement a VCS system is more like `git` handles things (to give you the idea).

THE BARE ESSENTIALS OF `git`

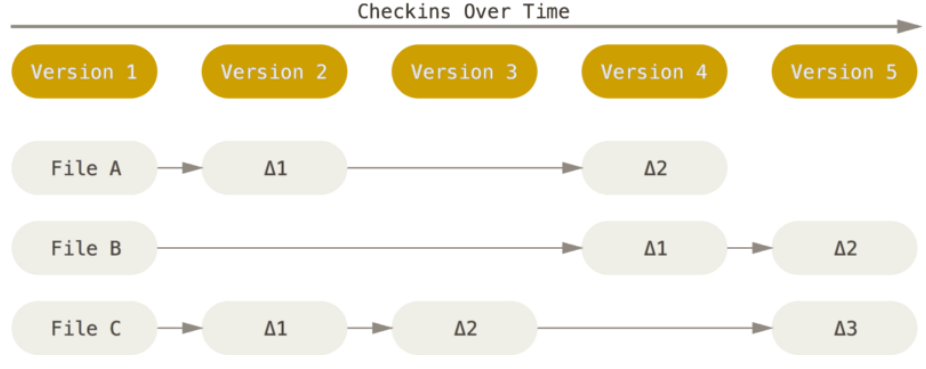


Storing data as a *stream* of snapshots, how `git` does it. Dashed lines indicate references to the previous version.

THE BARE ESSENTIALS OF `git`



git (each version is a snapshot of the full file system)



Most other VCS (each version records a set of differences between individual files)

THE BARE ESSENTIALS OF `git`

`git` *does not* reference its files by name!

- Everything in `git` is checksummed
- It uses the SHA-1 hashing algorithm for this
- The result looks something like this:
  `0568236076763052 8cc5188a81f88d5e64711608`
- Once the hash is computed, it is impossible (or extremely hard) to change (or corrupt) the underlying file system state
- It means that changes you commit ***are final*** *(especially once you push them to a remote to share your changes with others)*
- Before you share your changes, you can be messy and play around in your ***local*** repository as much as you like without breaking anything. This is where `git` stands out from all other VCSs!

THE BARE ESSENTIALS OF `git`

There are ***three*** file states in `git` that are *very important to understand*:

1. `modified` : you have changed the file but not committed to your local database
2. `staged` : you have marked a modified file to go into your next commit snapshot
3. `committed` : the data is safely stored in your local database (SHA-1 hash is computed)

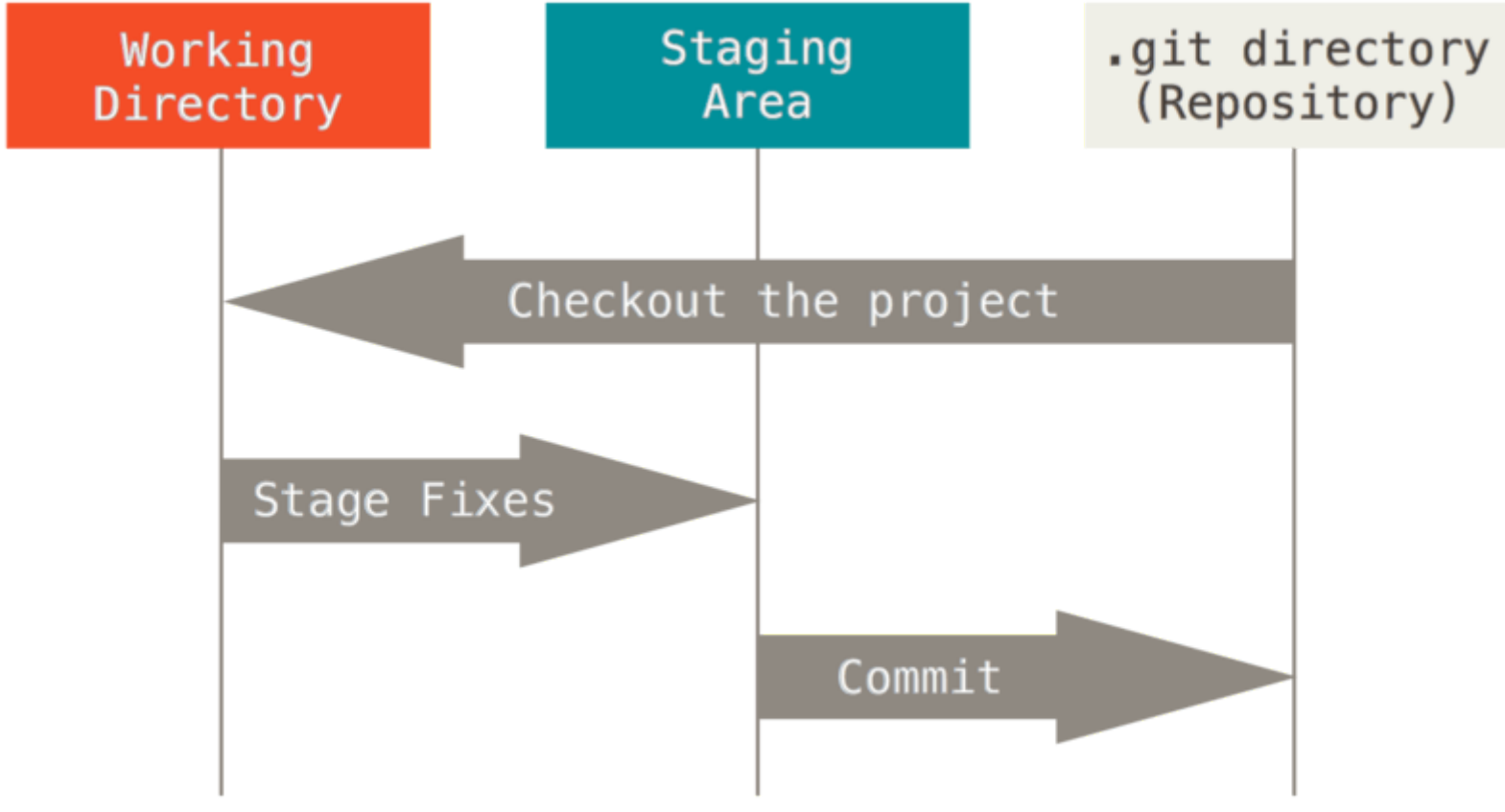
THE BARE ESSENTIALS OF `git`

This leads to three main sections of a `git` project:

1. The ***working tree*** where you modify files
   - Holds the files of a decompressed snapshot (`git` stores data using compression algorithms)
   - Here is where you play around and get creative
2. The ***staging area*** where you put your staged files (also referred to as ***index***)
   - The staging area is simply a file inside the `.git` directory called `index`
   - You *selectively* add changes to the `index` that you plan for the next commit
   - *You should add logical, small changes at a time*. If you are faced with a larger problem, divide it into smaller sub-problems and go step-by-step. **Why is this good advice?**
3. The ***`.git` directory*** where your commit *history* lives
   - This is where `git` stores metadata and the object database for your project. *When you remove it, you remove the VCS.*
   - It is the *heartbeat* of `git` and the data that is copied when you clone a project.

THE BARE ESSENTIALS OF `git`

This leads to three main sections of a `git` project:



The three main sections of `git` and how information flows.

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 1: obtain `.git` directory

- To start VCS using `git` you need a `.git` directory inside to *root* directory of your project:
  1. You can *initialize* a new one
  2. Or *clone* an existing project (you already did this in HW1)

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 1: obtain `.git` directory

- Initialize from scratch:

```
$ git init  # you must run this command in the root of your project
Initialized empty Git repository in /home/fabs/CS107/project/.git/
$ ls -a
.  ..  .git
```

- Clone a local project:

```
$ cd ..
$ git clone project cloned_project # local project from above
Cloning into 'cloned_project'...
warning: You appear to have cloned an empty repository.  # It's OK, I know
done.
```

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 1: obtain `.git` directory

- Initialize from scratch:

```
$ git init  # you must run this command in the root of your project
Initialized empty Git repository in /home/fabs/CS107/project/.git/
$ ls -a
.  ..  .git
```

- Clone from a URL: let's get the `git` project itself

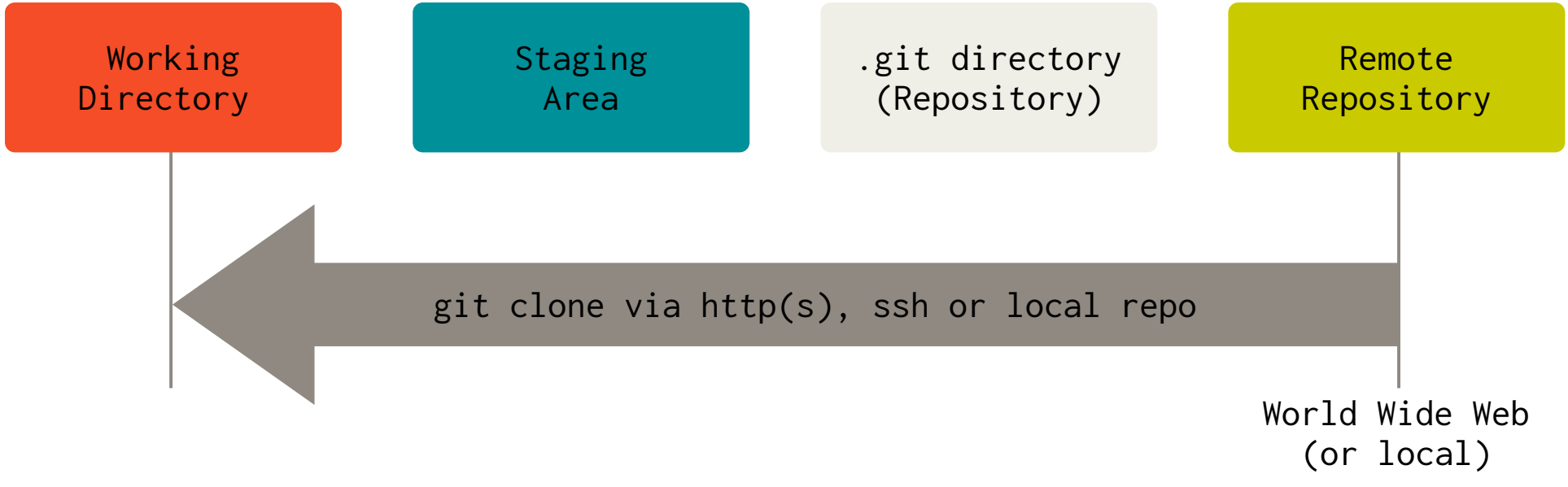
```
$ git clone https://git.kernel.org/pub/scm/git/git.git/
Cloning into 'git'...
remote: Enumerating objects: 8907, done.
remote: Counting objects: 100% (8907/8907), done.
remote: Compressing objects: 100% (789/789), done.
remote: Total 312519 (delta 8401), reused 8434 (delta 8117), pack-reused 303612
Receiving objects: 100% (312519/312519), 101.52 MiB | 6.52 MiB/s, done.
Resolving deltas: 100% (233246/233246), done.
```

- Getting help: `git help init`

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 1: obtain `.git` directory

When you clone a repository (usually from the web):



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 2: check your status (we are in the empty project of step 1)

```
$ git status  # inside: /home/fabs/CS107/project/
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$ ls .git  # no index yet...
config  HEAD  hooks  objects  refs
```

THE BARE ESSENTIALS OF `git`

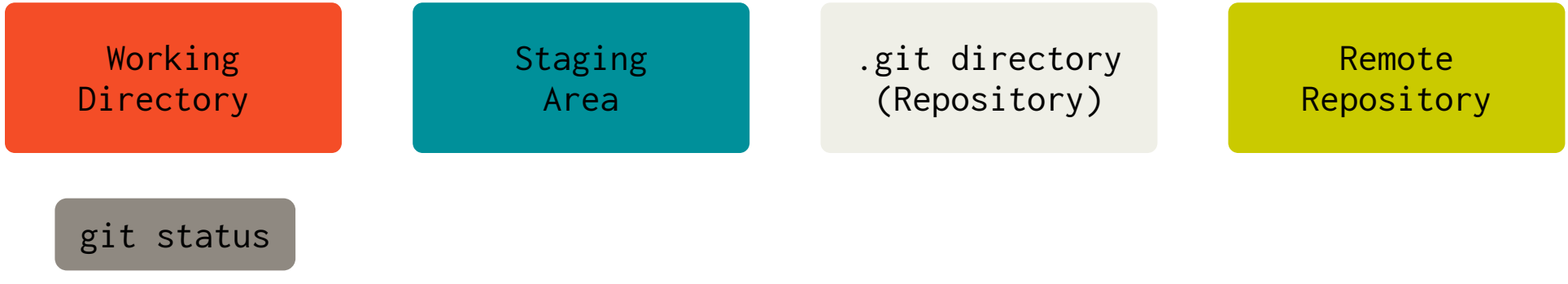
[Step-by-step example] Step 2: check your status (we are in the empty project of step 1)

```
$ git status  # inside: /home/fabs/CS107/project/
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
$ ls .git  # no index yet...
config  HEAD  hooks  objects  refs
$ echo "Hello Git!" > file
$ git status
On branch master

No commits yet

Untracked files:  # files that git is not aware of at this point
  (use "git add ..." to include in what will be committed)
        file      # we have just created this file

nothing added to commit but untracked files present (use "git add" to track)
```

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 2: check your status (we are in the empty project of step 1)

The status is checked in the working directory:



THE BARE ESSENTIALS OF `git`

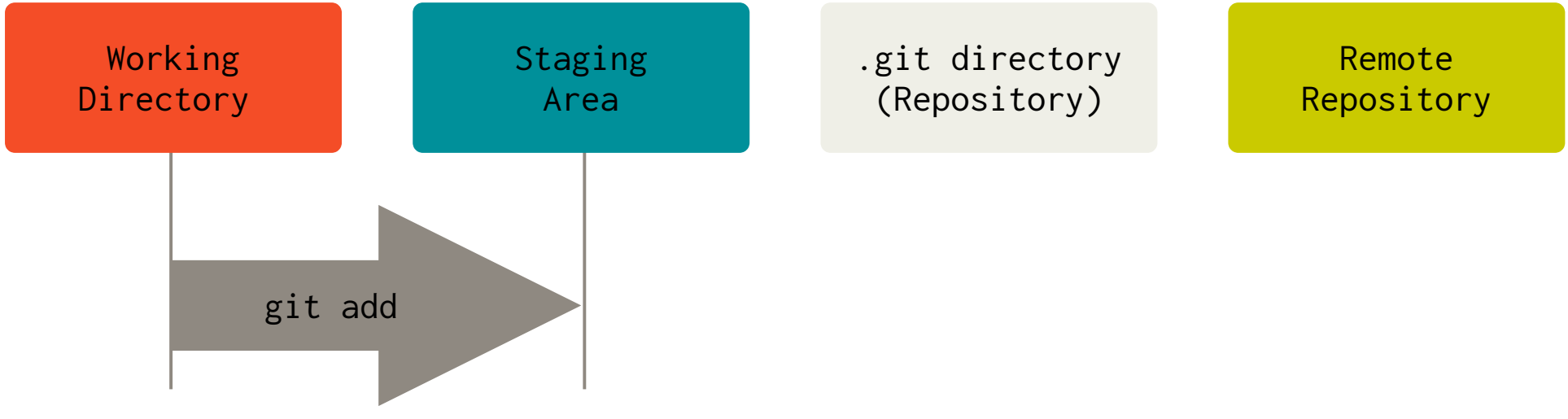
[Step-by-step example] Step 3: add new or modified files to the index

```
$ git add file  # inside: /home/fabs/CS107/project/
$ git status
On branch master

No commits yet

Changes to be committed: # now git is aware of the file...
  (use "git rm --cached ..." to unstage)
        new file:   file

$ ls .git  # ...and we have an index file (a.k.a staging area)
config  HEAD  hooks  index  objects  refs
```

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 3: add new or modified files to the index

New or changed files are added to the index (what you plan to commit next):



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 4: Commit staged changes to the repository:

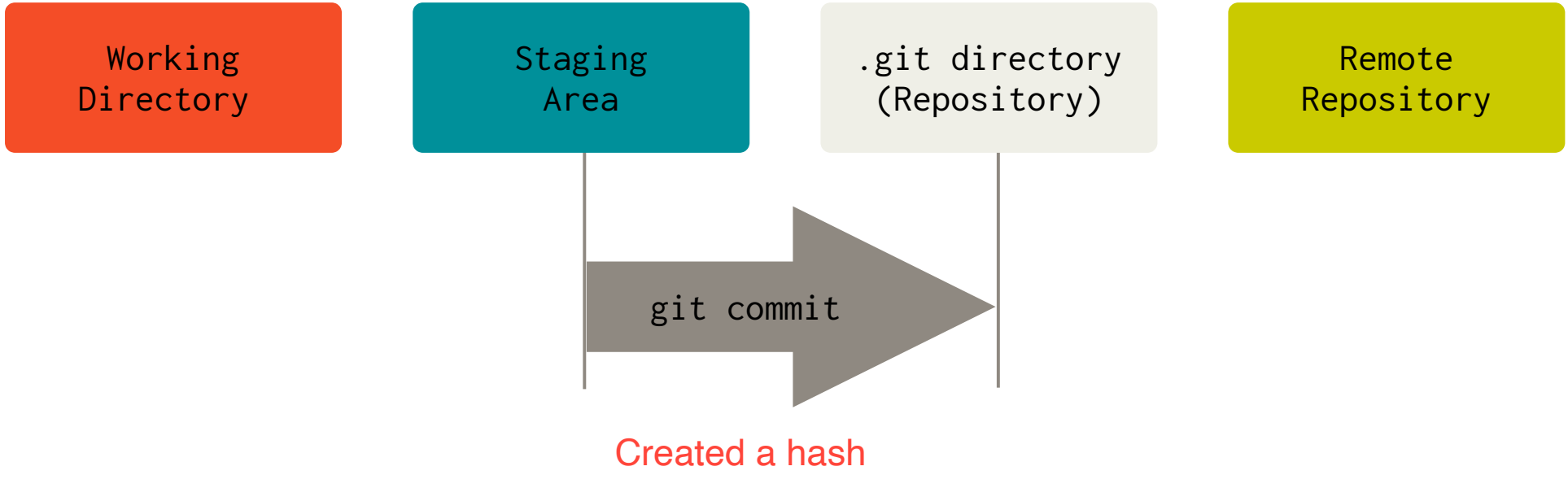
```
$ git commit -m "My short commit message"
[master (root-commit) 5e8adae] My short commit message
 1 file changed, 1 insertion(+)
 create mode 100644 file
```

- The `-m <message string>` option allows for a short commit messages on the command line. Useful if only few changes have been committed
- If you run `git commit` only, your editor specified in `~/.gitconfig` or the `GIT_EDITOR` environment variable will open up.
  See `git help commit` for more details.

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 4: Commit staged changes to the repository:

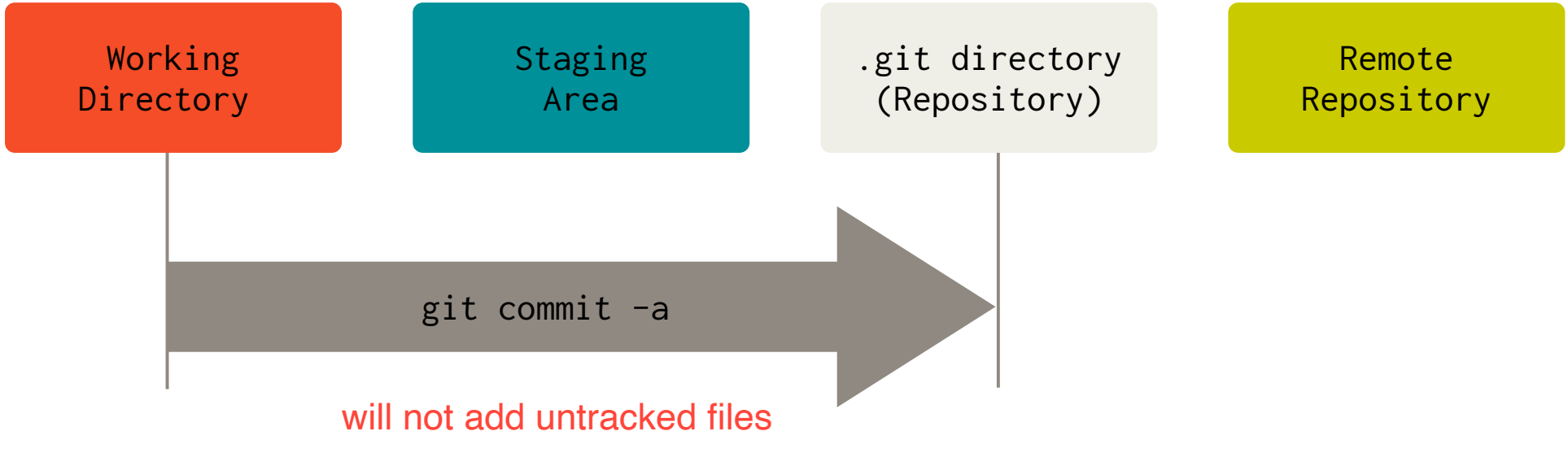
Staged changes are committed to the repository:



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 4: Commit staged changes to the repository:

Shortcut to add all *modified* or *deleted* (but not new) files in the working directory and commit right away:



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 5: Push commits to the repository:

```
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 264 bytes | 264.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To <remote path or URL>
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'
```

- The option `-u` is only needed if you push for the first time and you would like `git` to know that subsequent pushes from the local `master` branch are meant for the `origin/master` branch on the remote called `origin` (this is the default name for a remote in `git`)

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 5: Push commits to the repository:

Pushing commits to a remote repository:

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 6a: Synchronizing with the remote:

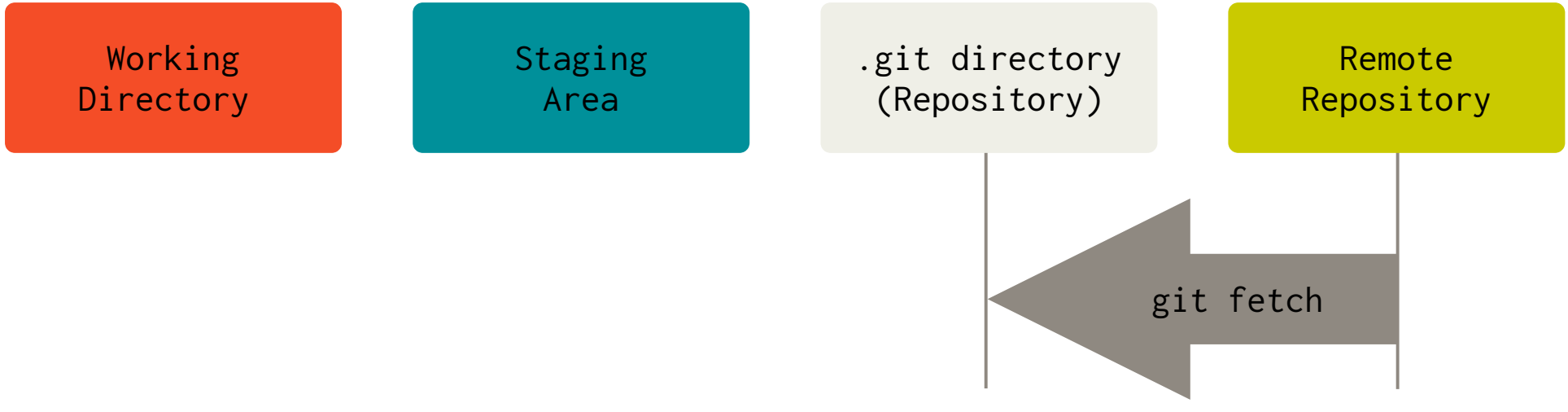
```
$ git fetch
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 247 bytes | 247.00 KiB/s, done.
From ../project_remote
   5e8adae..7b53cec  master     -> origin/master
```

- You only need `git clone` at the very beginning to get the `.git` directory
- Once you have a `.git` directory, you obtain new commits from the remote using `git fetch` (or `git pull`)

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 6a: Synchronizing with the remote:

Obtaining new commits from the remote:



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 6b: Synchronizing with the remote:

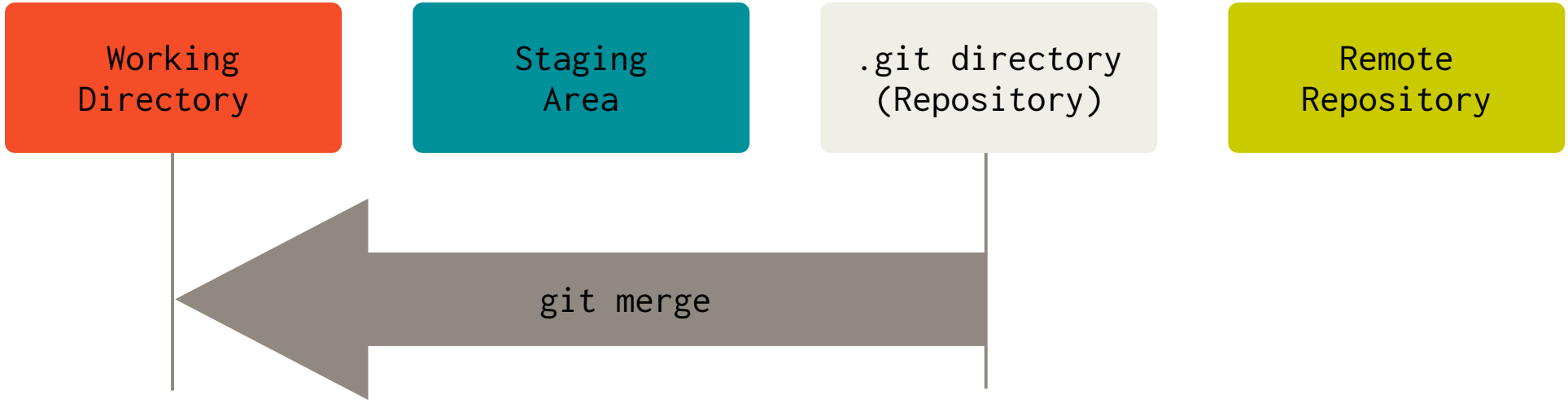
```
$ git merge
Updating 5e8adae..7b53cec
Fast-forward
 file | 1 +
 1 file changed, 1 insertion(+)
```

- To update your *working directory*, you need to merge the commits you just *fetched*

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 6b: Synchronizing with the remote:

Merging new commits from the local `.git` repository:



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 6c: Synchronizing with the remote:

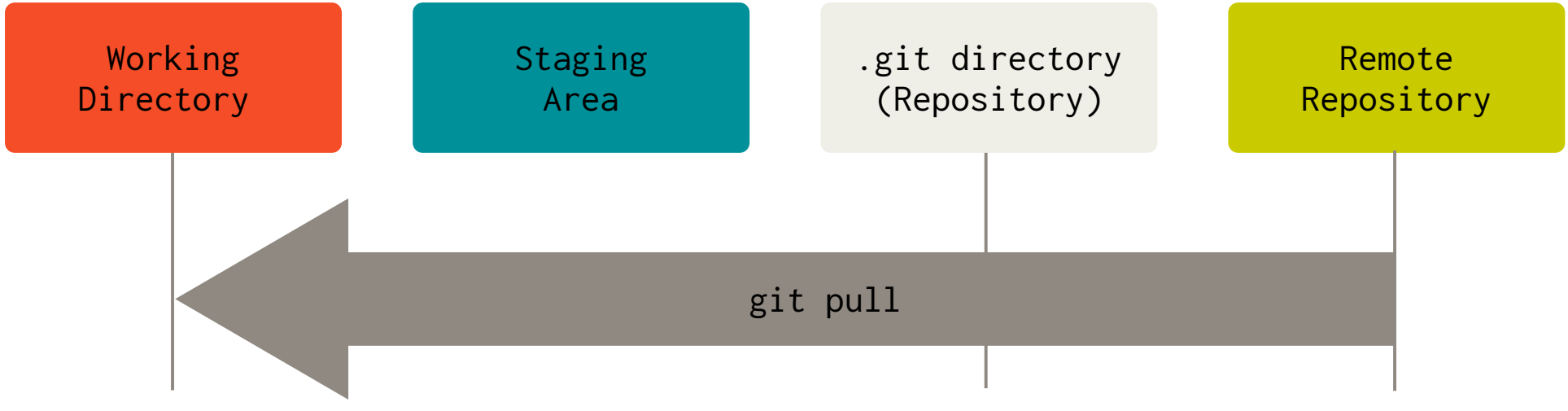
```
$ git pull  # combine git fetch and git merge in one go
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 248 bytes | 248.00 KiB/s, done.
From ../project_remote
   7b53cec..6a2d2af  master     -> origin/master
Updating 7b53cec..6a2d2af
Fast-forward
 file | 1 +
 1 file changed, 1 insertion(+)
```

- Combine `git fetch` and `git merge` in one go with `git pull`

THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 6c: Synchronizing with the remote:

Pulling new commits from the remote:



THE BARE ESSENTIALS OF `git`

[Step-by-step example] Step 7: See the log of commits (contributed by you and others):

```
$ git log
commit 7b53cec71d375b4e570af1113d5a76059de3c1d1 (HEAD -> master, origin/master)
Author: Fabian Wermelinger <fabianw@seas.harvard.edu>
Date:   Fri Sep 3 20:13:33 2021 -0400

    Quick fix

commit 5e8adae9f07b34fca41334b7997750e13bbe6c92
Author: Fabian Wermelinger <fabianw@seas.harvard.edu>
Date:   Fri Sep 3 19:20:42 2021 -0400

    My short commit message

    You can be more verbose in the body of the commit.
```

Or compact in one line per commit:

```
$ git log --oneline
7b53cec (HEAD -> master, origin/master) Quick fix
5e8adae My short commit message
```

HOW OFTEN SHOULD YOU COMMIT CHANGES?

- ***Remember:*** on your local repo, you can get messy and clean up later
- If you are working on something complex, it may make sense to commit very frequently, such that you can keep track of the impact of your changes (micro commits)
- You can always ***reorder*** and ***combine*** local commits *(before you push them to a remote where others have access too)*
- Committing once or twice a day is too few. Your commits will be bulky and it will be difficult to ***bisect*** if you introduced a bug...
- Final commits that you intend to push should *always build the code correctly* (if you work with a compiled language) and *pass all your unit tests*. If you do not take care of this and you share your commits with others, then they will not be very pleased with you!

INTERACTIVE `git rebase` DEMO

- When you have done some work in your local repository, you may want to clean up a bit before you push your changes to the remote.
- You can do this with an *interactive* rebase of your local commit history.
- The command for this is: `git rebase -i`
- You need an editor for the interactive changes.

interactive rebase

The demo `git` repo can be downloaded here:
https://harvard-iacs.github.io/2021-CS107/lectures/lecture3/

RECAP

- Introduction to version control systems
- Centralized and distributed approaches
- Bare essentials of `git`
- Interactive `git rebase` demo

REFERENCE

- Pro Git book by Scott Chacon and Ben Straub, Apress