SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 5

Fabian Wermelinger
Harvard University
CS107 / AC207

Thursday, September 15th 2022

LAST TIME

- Managing Jobs and processes in Linux, suspending and continuing execution.
- Introduction to version control systems
- Centralized and distributed approaches
- Essentials of Git
- Interactive `git rebase` demo

TODAY

Main topics: ***Version control systems (VCS), Managing repositories, Remote repositories, Branching***

Details:

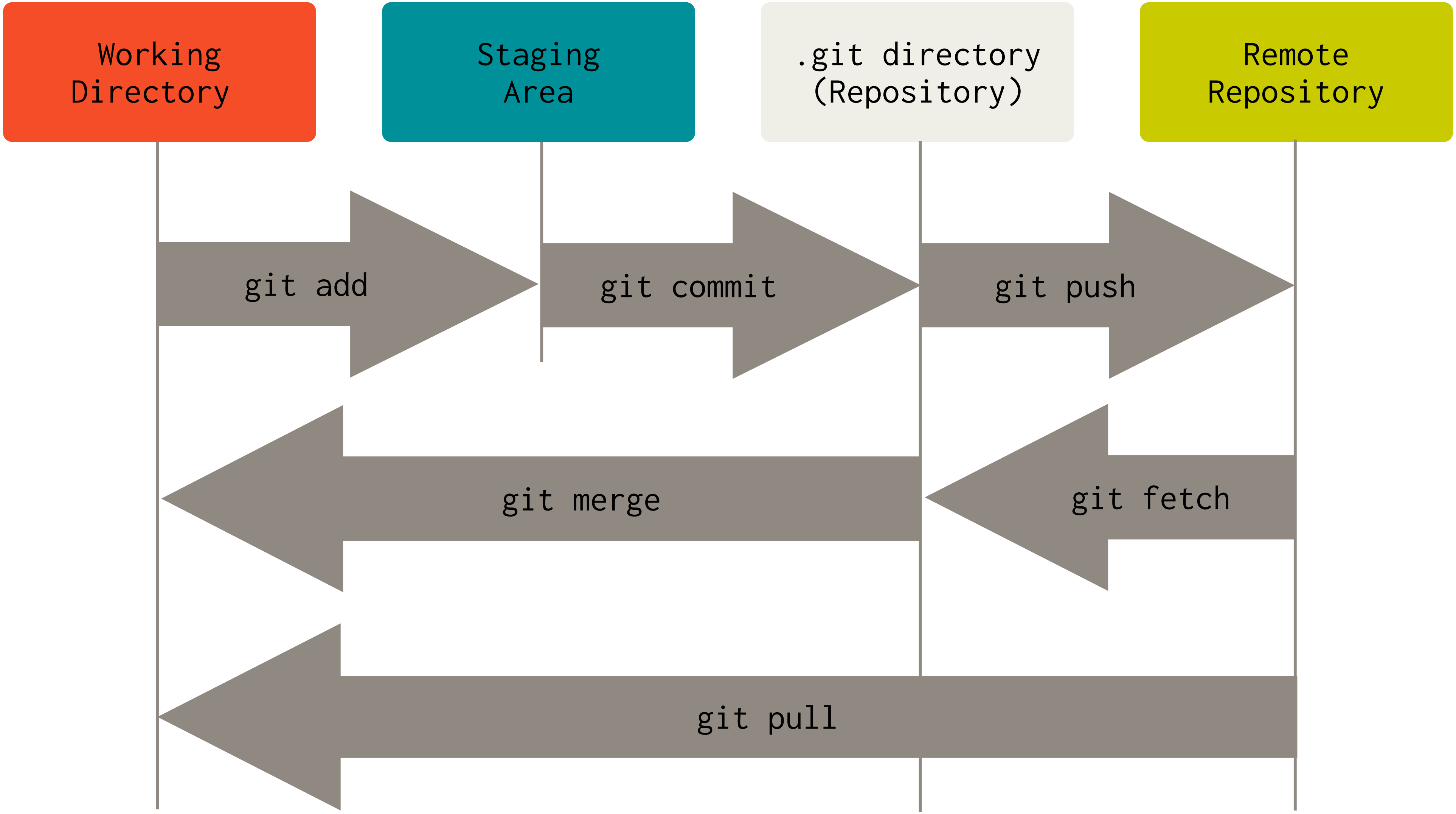
- More basic Git commands
- Repository maintenance
- Remote repositories
- Branching

AGENDA CHECK:

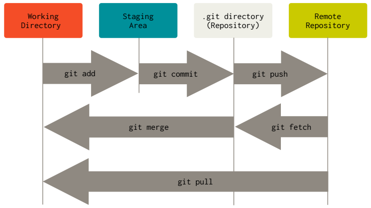
- Pair-programming 1 submission due tomorrow. *Submit the solution files in `lab/pp1` on your default branch*.

Disclaimer: Some content and figures in these slides are based on the free Pro Git book written by Scott Chacon and Ben Straub.

BASIC GIT COMMANDS YOU MUST KNOW BY HEART



BASIC GIT COMMANDS YOU MUST KNOW BY HEART



- `git add`: add new or modified files to the index (staging area in the `.git/index` file)

  Remark: you could use "`git add .`" to add *any* new or modified files in one go. This is ***bad practice*** because it may add files to the index that you did not intend to. Your colleagues will not be happy about this. Only lazy people do this.

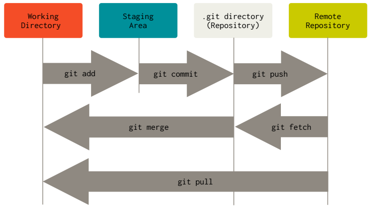
- `git commit`: commit the staged changes to the repo

  Remark: It is ***good practice*** to create small, well-arranged commits. You can always *rebase* if you think two (or more) small commits belong to one commit.

- `git push`: push commits to the upstream repository

  Remark: The upstream repository never has a working directory checked out. It only consists of the contents inside the `.git` directory. It can be on a remote location or locally (e.g. for backup purposes). See the `--bare` option of `git help init`.

BASIC GIT COMMANDS YOU MUST KNOW BY HEART



- `git fetch`: fetch new commits from the upstream repository (e.g. from your collaborators)

  Remark: This will only update your local `.git` repository, not your working directory.

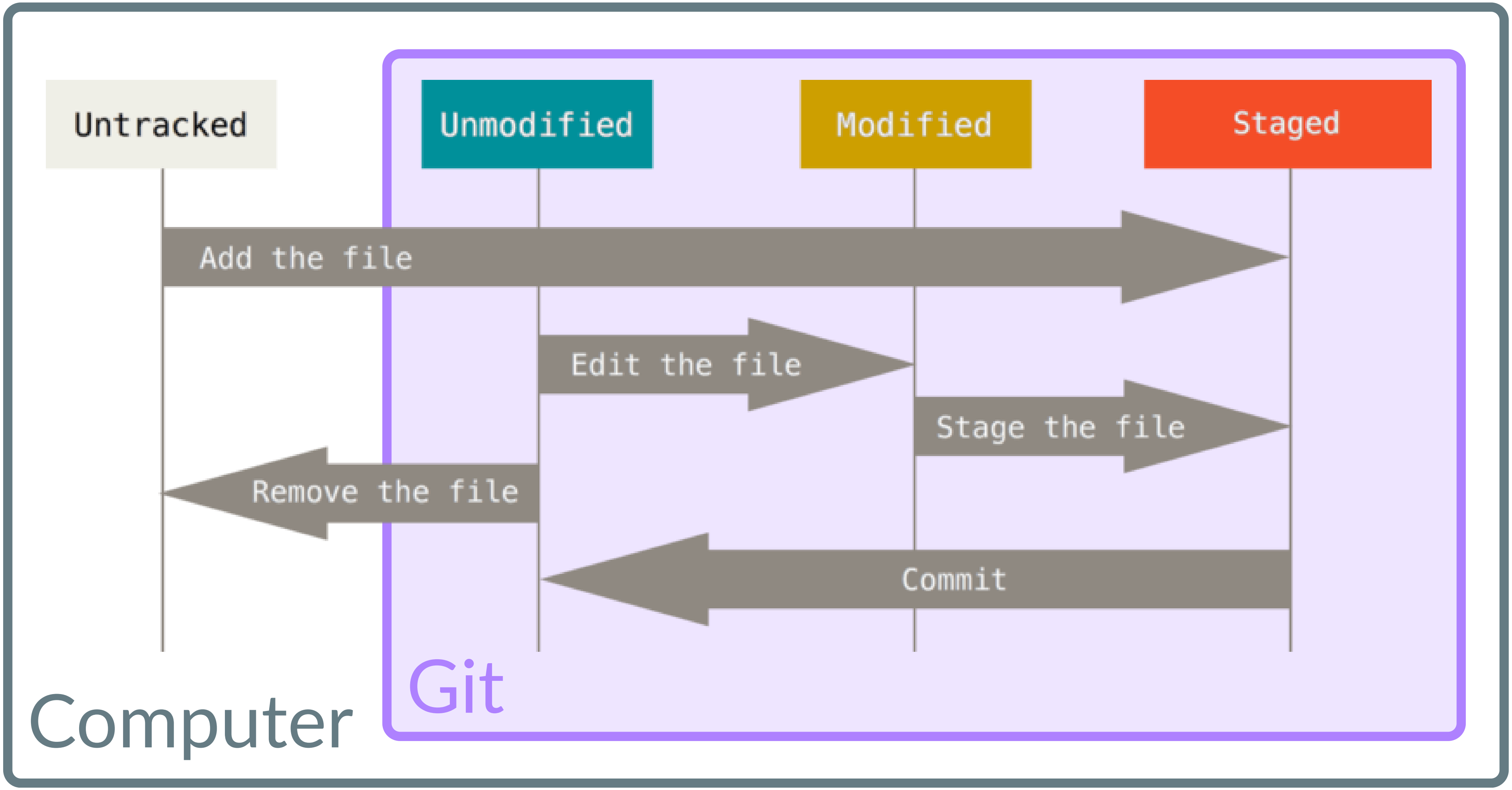
- `git merge`: update your working directory by merging new commits from your local repository (joining histories)

  Remark: By default this merges the tracked remote branch (`git fetch` first). You can merge any other branch you like (discussed later in this slide set).

- `git pull`: fetch commits from the upstream repository and merge them with the current working directory

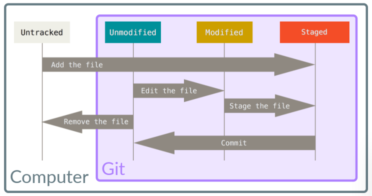
  Remark: This saves you some time as most often you want this behavior, rather than executing `git fetch` followed by `git merge`.

UNDERSTAND THE STATUS OF YOUR FILES



Staging area and ***index*** are synonyms.

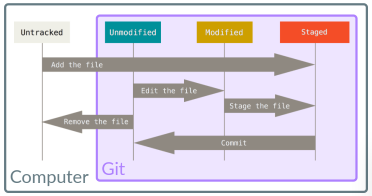
UNDERSTAND THE STATUS OF YOUR FILES



From *untracked* to *staged*:

```
$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
$ git add file
$ git status
On branch master

No commits yet

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

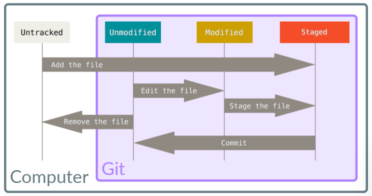
UNDERSTAND THE STATUS OF YOUR FILES



From ***staged*** to ***unmodified***: record the snapshot

```
$ git commit -m "Added untracked file"
[master (root-commit) 68581f7] Added untracked file
 1 file changed, 1 insertion(+)
 create mode 100644 file
$ git status
On branch master
nothing to commit, working tree clean
```

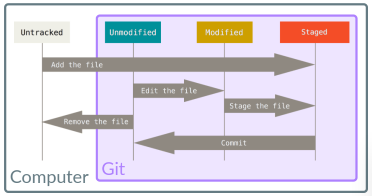
UNDERSTAND THE STATUS OF YOUR FILES



From *unmodified* to *modified*: edit the tracked file

```
$ echo 'Adding a new line of text' >>file
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file

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

UNDERSTAND THE STATUS OF YOUR FILES

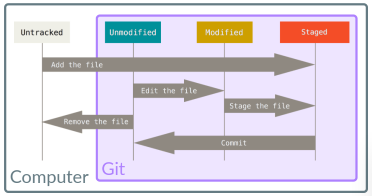


From *modified* to *staged*: add the file to the index

```
$ git add file
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file
```

- We can now run `git commit` again to go from staged to unmodified (by recording a new *snapshot* in the history)
- Instead of running `git add file` and then commit, we can combine these steps with `git commit -am <commit message>`
- "`git commit -a`" ***is not*** the same as "`git add .`" followed by `git commit` (the former only works with tracked files, the latter adds *untracked* files too!)

UNDERSTAND THE STATUS OF YOUR FILES

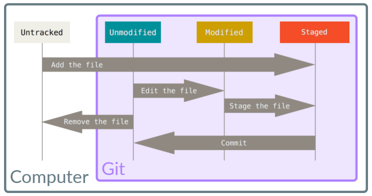


If you modify a ***staged*** file again:

```
$ echo "Modify a staged file" >> file
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file
$ git add file
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   file
```

- New modifications are separate from the ones you have staged already.
- If they belong in the same commit, then you need to run `git add file` *again* to add your new changes to the already staged changes!

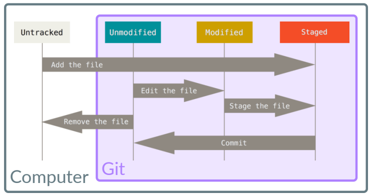
UNDERSTAND THE STATUS OF YOUR FILES



You can remove tracked files from the repository. Removing files is two-fold in Git (*see computer frame and Git frame in image on the left*):

1. Remove files from the `.git` repository only, keep them in your file system
2. Remove files from both, `.git` repository ***and*** your file system.

UNDERSTAND THE STATUS OF YOUR FILES

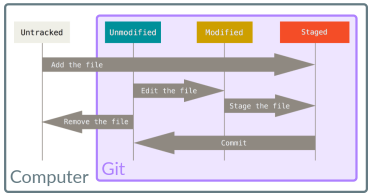


1. Remove files from the `.git` repository only, keep them in your file system

```
$ git ls-files  # list files that are known to git
file
$ git rm --cached file
rm 'file'
$ git ls-files  # no output = no files tracked
$ ls
file  # the file is still with us, but not under VCS anymore
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file

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

We still need to commit the changes, even when we delete files from the repository. ***Remember:*** Git thinks in terms of file systems, you must record a snapshot when you remove files too.

UNDERSTAND THE STATUS OF YOUR FILES

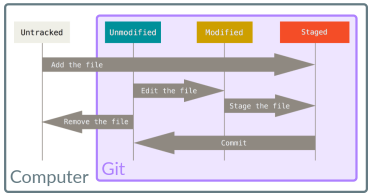


1. Remove files from the `.git` repository only, keep them in your file system
2. Remove files from both, `.git` repository ***and*** your file system.

```
$ git rm file
rm 'file'
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file
```

- This time the file is gone (Git does not report untracked files in the current directory)
- But fear not, we can still restore any removed files from the Git file system using any snapshot. The command is `git restore`
- Restoring files in Git can be a lifesaver in some situations.

UNDERSTAND THE STATUS OF YOUR FILES



- Note that Git uses the *same command name* to remove files as on the Linux command line itself: `rm`
- The same applies when you want to *rename files*. In that case you would use the `mv` command (move) in the Linux command line as well as for `git`: `git mv`

```
$ git mv file new_filename
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    file -> new_filename
```

- To move a file means the file system has changed. Git requires you to commit a snapshot for this action, as usual.

KNOW YOUR HISTORY

- Every commit you make in Git is recorded in the ***history***.
- The history contains a huge amount of information and obviously is important when you need to comprehend changes that were not committed by you.
- For that reason, every commit must be documented accurately.
- *It is not easy to write good and concise commit messages!*

KNOW YOUR HISTORY

Good practices for commit messages:

- The structure of a commit message:

```
Message subject: One single line, (should be) not more than 72 characters

Longer message body: The body provides more details if the commit
contains a complex change.  It can consist of multiple paragraphs,
formatted at 72 characters per line maximum (some projects are very
strict on these format requirements because the commit will go into the
history of the project and it should maintain a consistent format).
These format requirements are usually implemented (or can be configured
easily) in editors like vim or emacs when you write git commit messages.
You may omit the message body if your commit is small and the subject is
descriptive enough.

(The subject line above is actually 74 characters long...)
```

- ***Write concise message subjects!***

```
$ git log --oneline
9cb047b (HEAD -> master) Add license information in README.md    <- OK
b51859a (origin/master) Add tests             <- Not very descriptive!
2c1f77c Minor                                 <- This is bad!
5dfb982 Bug fix                               <- Also bad!
```

KNOW YOUR HISTORY

- You display the history using `git log`
- The structure of a Git history entry looks like this:

```
commit 72e96d44caf034fdad447eb40ff9cf001075bd0f
Author: Fabian Wermelinger <author@domain.net>
Date:   Mon Jun 21 18:38:39 2021 +0200

    Add src_field_ and dst_field_ for pointwise kernel inputs

    Memory layout in fields is more favorable for pointwise operations than
    pitched layout in labs.  This allows to test kernels that take a field
    (without ghost cells) as input source.
```

  - Commit identifier (SHA-1 hash)
  - Commit author/committer and date
  - Commit subject
  - Followed by commit body *(may be omitted if subject contains sufficient information)*

KNOW YOUR HISTORY

Searching the Git history:

- Often you need to search the history for specific keywords, commits or the commit author. Git uses a `grep`-style search engine.
- You can specify the `--grep=<pattern>` option to search for a regex pattern. This searches *log messages (subject and body)*
- You can specify the `--author=<pattern>` option to search for a particular author/committer using a regex pattern. This only searches author information but not commit messages.
- If you use the `--grep` option multiple times, any pattern may match. If you want that *all patterns must match* pass the `--all-match` option.

KNOW YOUR HISTORY

Formatting the output:

- You can change the format of how displays the history log.
- Use `git log --pretty=oneline` to display the commits in compact form (this option also exists as `--oneline` because it is used often). If you want a lot of information, you can use `--pretty=fuller` instead. See `git help log` for docs.
- Your Git installation also ships with a graphical tool that you may use to explore the history. You can use the `gitk` tool or `git gui`
- You can use the `[alias]` section in your `~/.gitconfig` to define multiple output layouts.

IGNORE DATA YOU DO NOT WANT TO TRACK!

- In Git you can use one or many `.gitignore` files to *ignore* files you do not want to track (it is a *hidden* file).
- Notoriously annoying files are editor backup files (`*~`, `*.bak`, `*.swp`), object files of compiled languages like C or C++ (`*.o`) and `.DS_Store` or the `__MACOSX__` directory.
- You add these patterns one per line in `.gitignore`:

```
# you can use comments too!
__pycache__/  # this ignores a whole directory
*.bak         # name of backup files
*~            # some editors create backup files ending with '~'
*.pyc         # pre-compiled Python bytecode (not portable)
```

- Use "`!`" for ***negation***:

```
*     # ignore everything (wildcard expands to anything)
!*.py # except any file with the .py suffix
```

COMMENTS ABOUT `.gitignore`

- **It is essential that you keep your repository clean**, the `.gitignore` file is the key to a clean repository.
- You usually have one in the root directory and possibly others in more specific sub-directories of your project.
- Often they have specific entries for the programming language you are using. E.g. for Python you want to ignore the `__pycache__` directories.
- GitHub offers some templates for this file at the time when you create a new repository, have a look at them to get an idea. (I often prefer to create them from scratch and extend them on the fly.)

COMMENTS ABOUT `.gitignore`

- Git is great for VCS of *text* files
- It can handle binary files but they are more difficult to track and compress efficiently (recall the *blob*). This often leads to an increased storage footprint for your `.git` repository.
- Ignore such files in your `.gitignore` files. E.g. for PDF files:

  ```
  *.pdf # ignore all pdf's
  ```

  If you require an exception, you can force add the file to the index using the `-f` or `--force` option:

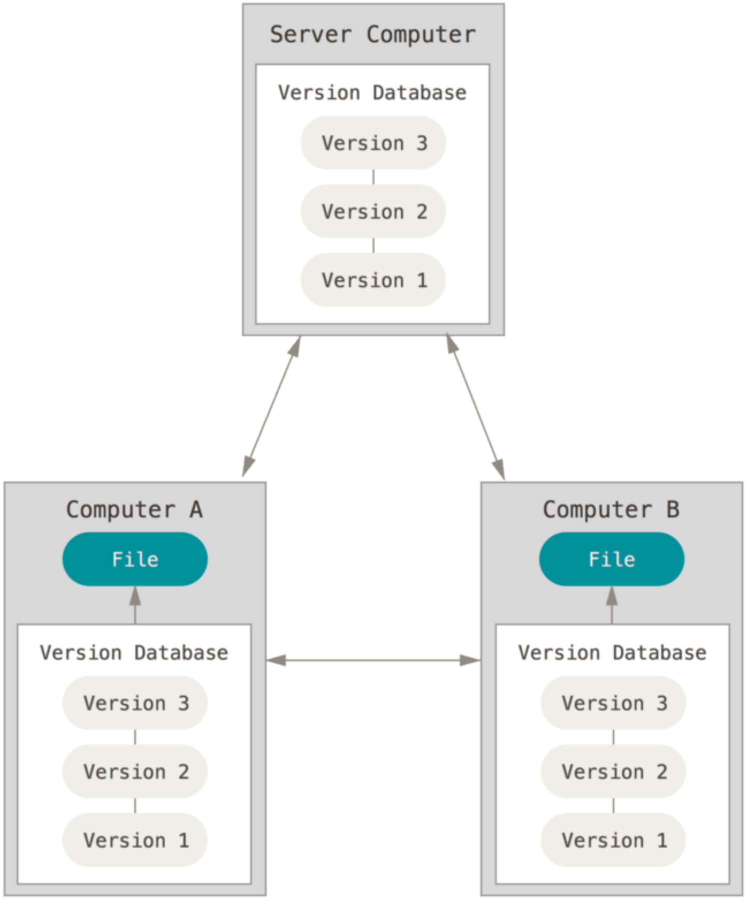
  ```
  $ git add --force important.pdf
  ```

GIT HELP

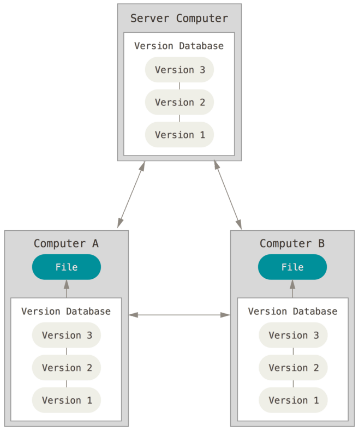
- All commands in Git have proper manual pages.
- You can get them in two ways:
  1. Via `git`: `git help commit`
  2. Via `man`: `man git-commit`

REMOTE REPOSITORIES

Recall distributed VCS:

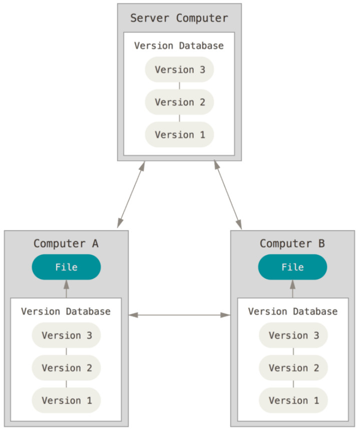


REMOTE REPOSITORIES



- The *"server computer"* is called a *remote*.
- It can be a server from GitHub, for example, but *it can also be local on your computer*. *Git does not really care about the "where"*.
- So, the term *"remote"* does not necessarily imply that it is some place else on the internet, only that the remote repository is *somewhere* else and is usually used as a *hub* only.
- All that a remote repository needs is the content of the `.git` directory, such a repository is called a ***bare*** repository. *You cannot checkout a working tree.*
- You can list the remotes with `git remote show`.

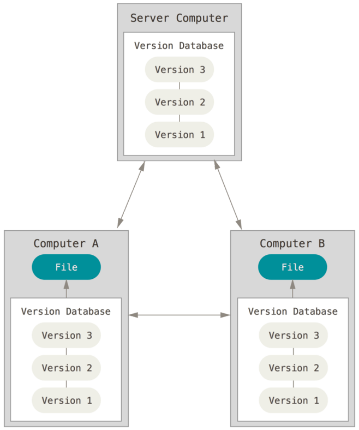
REMOTE REPOSITORIES



We can easily simulate this situation:

```
rm -rf git
mkdir -p git/remote

# initialize remote (bare repository)
(cd git/remote && git init --bare)

# initialize A
mkdir -p git/A
cd git/A
git init
git config user.name 'Developer A'
git config user.email 'A@domain.org'
git branch -M main

# setup the remote and create content
git remote add origin ../remote # no URL this time
echo 'Initial' >file
git add file
git commit -m 'Initial'
git push -u origin main

# B clones
cd .. # inside `git` directory
git clone remote B
cd B # inside repository B
git config user.name 'Developer B'
git config user.email 'B@domain.org'
git config branch.main.rebase 'false'
# default branch name already defined by A
```

REMOTE REPOSITORIES



We can easily simulate this situation:

- *Merge conflicts* are not uncommon and are the trickiest part in Git. They are hard to avoid in distributed VCS.
- To avoid merge conflicts further, you should aim committing small changes (micro-commits). The resolution process will be much more clear if the conflicting changes you have to deal are small.
- Git also offers a tool to resolve merge conflicts (*part of pair-programming 3 next week)*:
  `git help mergetool`
- `vim` users checkout this Git plugin: https://github.com/tpope/vim-fugitive

SUMMARY REMOTE REPOSITORIES

- You can add as many remotes as you like.
- If you do not setup a tracking branch for `git push`, then you must be explicit and tell Git which remote you to use and which branch to push (same is true for `git pull`).

How to access remote servers:

- On GitHub you can choose to use `https` or `ssh` (prefer `ssh`) to communicate with a remote.
- If you use `https` you now must create a *token* in "Settings/Developer Settings".
- For `ssh` you can generate an RSA key using `ssh-keygen -t rsa -b 4096` and upload the *public* key to GitHub (you should have done this already).

BRANCHES

- We have encountered branches already but not said much about them up to now.
- In your GitHub repo, `main` or `master` *are branches*. A *branch* is similar to a linked list of *commits* and it is referenced by the most recent commit (the pointer `HEAD` in your active branch).
- Branches are your ***main tool*** for development. Whenever you think about testing something out, the first thing you do is create a new branch. You can just *discard* the branch if it does not work.
- Historically, branching is an expensive task in VCS, not in Git!
- ***Recall:*** blobs, trees and commits is what Git cares about. The reference of a branch simply is a commit reference *(a file system snapshot)*

```
$ cat .git/refs/heads/main  # 'heads' means heads of branches
d5278fbd2931b75e9f41ef031448fa1b2696fce4
```

BRANCHES

- Assume you have a new repository and you just created the initial commit `A`:

  ```
  A main
  ```

  > In the following the *pointer* denoted `main` points to the *head commit* of the branch `main`. Currently, this is commit `A`.

- Now suppose we make two more commits `B` and `C`. The pointer that describes the `main` branch moves along:

  ```
  A---B---C main
  ```

- At this point in your development process, you notice a strange behavior of your code and you suspect that a bug has been introduced. ***How to proceed now?***

BRANCHES

- You can continue on `main` but this *is not a good idea*. Fixing bugs requires you to throw things around. *Create a new branch `bugfix1`:*

```
$ git switch -c bugfix1   # the -c option creates the branch if it does not exist
```

Note: in older versions of `git` a new branch was checked out like this

```
$ git checkout -b bugfix1   # the -b option creates the branch if it does not exist
```

The reason this is confusing is because `git checkout` has *dual* meaning:

1. It can checkout branches
2. It can checkout individual files and restore their content

Newer versions of `git` split these tasks by introducing two new commands:

1. `git switch`: switch branches
2. `git restore`: restore files

- Our revision timeline now looks like this:

```
A---B---C main   # branch point is C
         \
          bugfix1  # branch point is C, active branch (what HEAD points to)
```

BRANCHES

- Our revision timeline now looks like this:

```
A---B---C main    # branch point is C
         \
          bugfix1  # branch point is C, active branch (what HEAD points to)
```

> There is a special pointer in Git called HEAD. It always points to the *currently active branch*.

- Now we do some work to fix this bug. Assume the next two commits `D` and `E` implement these fixes:

```
A---B---C main   # this is the active branch now
         \
          D---E bugfix1  # this branch contains the bug fixing code
```

  We also switched back to the `main` branch with `git switch main`.

- ***Which commit reference will HEAD now point to?***

BRANCHES

- We have *tested* our changes on the `bugfix1` branch and things work as expected. We switched back to the `main` branch as we would like to ***merge*** the history of `bugfix1` into `main`.

```
A---B---C main   # this is the active branch now
         \
          D---E bugfix1   # this branch contains the bug fixing code
```

- Because there are *no new commits* on `main` since we branched off, the merge is trivial. Git has two options:

1. Fold `bugfix1` and `main` together (*fast-forward merge*)
2. Create a *merge commit* which joins `bugfix1` and `main` in a common commit.

BRANCHES

- Situation *before* merge:

```
A---B---C main  # this is the active branch now
         \
          D---E bugfix1  # this branch contains the bug fixing code
```

- ***Fast-Forward merge:*** this is the default that Git assumes. Assume you are on the `main` branch, the command for this merge is `git merge [--ff] bugfix1` *(the fast-forward option `--ff` is implied if not given)*
  After the fast-forward merge your history looks like this:

```
A---B---C---D---E main
                 \
                  bugfix1  # this branch is now dangling
```

- The `bugfix1` branch is now *fully* merged in `main`. It is no longer needed and good practice to remove it: `git branch -d bugfix1`

```
A---B---C---D---E main
```

BRANCHES

- Situation *before* merge:

```
A---B---C main  # this is the active branch now
         \
          D---E bugfix1  # this branch contains the bug fixing code
```

- ***Merge with new commit:*** this type of merge creates a common commit for the merge (it will have 2 parents!):

```
git merge --no-ff bugfix1
```

  After creating a merge commit your history looks like this:

```
A---B---C-------F main # F is called a merge commit
         \     /
          D---E bugfix1  # this branch is now dangling
```

- Same rule for cleaning: `git branch -d bugfix1`

```
A---B---C-------F main
         \     /
          D---E
```

BRANCHES

Compare the difference of the two approaches:

Fast-Forward:

```
$ git log --oneline --graph
* 57f4883 (HEAD -> main) Commit E
* d5278fb Commit D
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

With merge-commit:

```
$ git log --oneline --graph
*   4466977 (HEAD -> main) Merge branch
|\                         (Commit F)
| * 57f4883 Commit E
| * d5278fb Commit D
|/
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

The difference:

Linear history:

```
A---B---C---D---E main
```

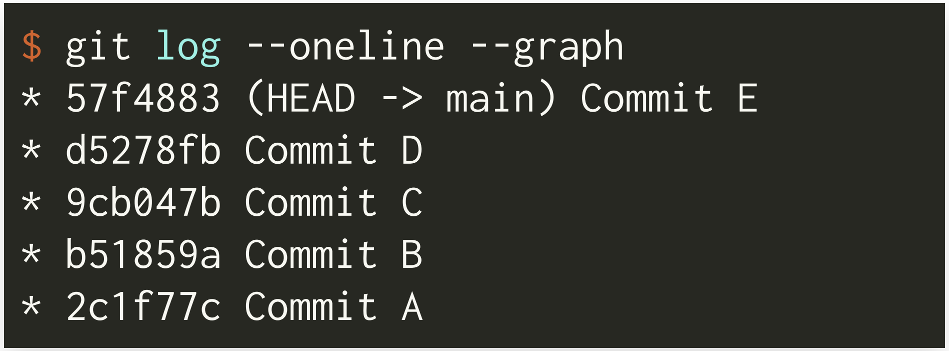
Non-linear history:

```
A---B---C-------F main
         \     /
          D---E
```

BRANCHES

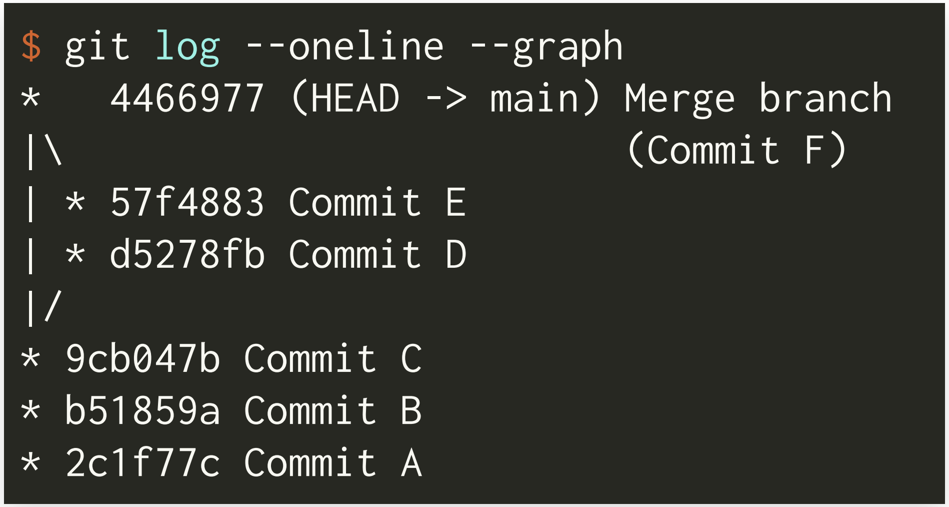
Compare the difference of the two approaches:

Fast-Forward:



```
$ git log --oneline --graph
* 57f4883 (HEAD -> main) Commit E
* d5278fb Commit D
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

With merge-commit:



```
$ git log --oneline --graph
*   4466977 (HEAD -> main) Merge branch
|\                         (Commit F)
| * 57f4883 Commit E
| * d5278fb Commit D
|/
* 9cb047b Commit C
* b51859a Commit B
* 2c1f77c Commit A
```

- Some people argue that creating merge commits adds *noise* to your history (technically they are not needed)
- Merge commits preserve your branching history, which may be useful for a better understanding of the development process.
- Some projects have requirements for how commits are merged.

STASHING CHANGES WITHOUT COMMITTING

Assume you find yourself in this situation:

```
A---B---C main
         \
          D-* bugfix1 # a bugfix branch, active branch ('*' means modified files)
```

Common scenario: you stop work on the `bugfix1` branch temporarily and need to switch to some other branch (say `main`). Your work on `bugfix1` is not ready to be committed yet.

```
$ git status
On branch bugfix1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file

no changes added to commit (use "git add" and/or "git commit -a")
$ git switch main
error: Your local changes to the following files would be overwritten by checkout:
        file
Please commit your changes or stash them before you switch branches.
Aborting
```

STASHING CHANGES WITHOUT COMMITTING

Assume you find yourself in this situation:

```
A---B---C main
         \
          D-* bugfix1 # a bugfix branch, active branch ('*' means modified files)
```

Common scenario: you stop work on the `bugfix1` branch temporarily and need to switch to some other branch (say `main`). Your work on `bugfix1` is not ready to be committed yet.

If committing the changes is too early, you can use `git stash` to *temporarily* stash your changes away:

```
$ git stash  # push current work on top of stash stack
Saved working directory and index state WIP on bugfix1: b955cbe Adding new file on bugfix1
$ git stash list  # list all stashed work, WIP means Work In Progress
stash@{0}: WIP on bugfix1: b955cbe Adding new file on bugfix1
$ git status  # working tree is clean now
On branch bugfix1
nothing to commit, working tree clean
$ git switch main  # do some other work on main
Switched to branch 'main'
```

STASHING CHANGES WITHOUT COMMITTING

Assume you find yourself in this situation:

```
A---B---C main
         \
          D-* bugfix1 # a bugfix branch, active branch ('*' means modified files)
```

Common scenario: you stop work on the `bugfix1` branch temporarily and need to switch to some other branch (say `main`). Your work on `bugfix1` is not ready to be committed yet.

If committing the changes is too early, you can use `git stash` to *temporarily* stash your changes away:

```
$ git switch bugfix1  # when done return to your bugfix1 branch
Switched to branch 'bugfix1'
$ git stash pop  # apply your last stashed changes, i.e. stash@{0}
On branch bugfix1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (e3e552a02d84049a314e77edcb34dae0987ef145)
```

REBASE A HISTORY TO MAINTAIN LINEARITY

Lets return to our previous state but now *we have a collaborator who did work* on `main` in the meantime (the commit labels `A`, `B`,... are only symbolic, don't take them literally):

```
A---B---C---D---E main   # work has advanced on this branch
         \
          F---G bugfix1    # this branch contains the bug fix, active branch
```

Can you apply a fast-forward merge strategy in this case?

- ***You can not!*** Remember, once a history is recorded by computing the SHA-1 hash, we can not change it anymore.
- There are two options:

  1. Merge via a *merge commit* (as before)
  2. ***If*** `bugfix1` is a branch that only exists in your *local* `.git` repository, we can rebase and therefore *rewrite the local history* (nobody has seen your local history yet). ***This a powerful feature of Git***.

REBASE A HISTORY TO MAINTAIN LINEARITY

```
A---B---C---D---E main  # work has advanced on this branch
         \
          F---G bugfix1   # this branch contains the bug fix, active branch
```

- ***Merge via merge commit:*** same as in the previous case where work on `main` did not advance:
  1. `git switch main` (change to the target branch)
  2. `git merge bugfix1`

     ```
     A---B---C---D---E---H main  # this is the active branch now
              \         /
               F-------G bugfix1   # this branch contains the bug fix, now dangling
     ```

  3. `git branch -d bugfix1` (clean up)

REBASE A HISTORY TO MAINTAIN LINEARITY

```
A---B---C---D---E main  # work has advanced on this branch
         \
          F---G bugfix1   # this branch contains the bug fix, active branch
```

- ***Rebase and merge:*** here we first *rebase* our `bugfix1` branch *onto* the advanced `main` branch and then use a fast-forward merge to *linearize* the history (we start with the state shown above):
  1. `git rebase main` ***(rewriting history here, i.e. new commits are created!)***

     ```
     A---B---C---D---E main  # work has advanced on this branch
                      \
                       F'---G' bugfix1  # after rebase, active branch
     ```

     - Commits `F'` and `G'` have a ***different*** SHA-1 than `F` and `G`, therefore, history is rewritten!
     - Their *time stamp* remains the same but parents change.
  2. `git switch main && git merge bugfix1`

     ```
     A---B---C---D---E---F'---G' main  # rebased history
                               \
                                bugfix1   # now dangling
     ```

  3. `git branch -d bugfix1` (clean up)

REBASE A HISTORY TO MAINTAIN LINEARITY

- `git rebase` unwinds commits and re-applies them on top of another commit. Naturally, this changes your history. It is a powerful tool for *local* history transformations.
- Rebased histories can have commit time stamps that are *not* in chronological order, but allow you to maintain a linear history.
- Rebasing allows to *linearize* your history
- Again, some projects are very strict about how you have to maintain the history of the project. Be sure to check them out before collaborating.

You can put yourself in a bad light if you rebase a history and (forcefully!) push it to a remote where others can pull from as well. Git will not allow you to do this by default, but you can force it with `git push --force`. It will *invalidate* the history in all your collaborators' local repositories. You can always rewrite history ***locally*** or use a forced push *iff* you are *the only one* working with the (remote) branch.

RECAP

- More basic Git commands → use `git status` often!
- Remote repositories → differences between a normal `.git` repository and a *bare* `.git` repository.
- Branching in Git → fast-forward merges and rebasing, linear and non-linear histories.

Further reading:

- Chapters 3, 4, 5, 6, and 7 in Scott Chacon and Ben Straub, *"Pro Git book"*, open access https://git-scm.com/book/en/v2
- Git command reference