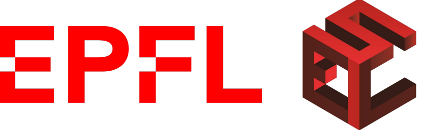


Introduction to git part 3

EPFL - Embedded Systems Laboratory

2025

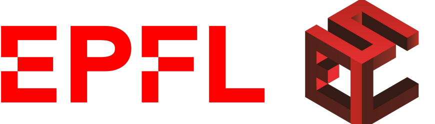
Introduction to git
introduction and basic use

Working alone
for any project you have

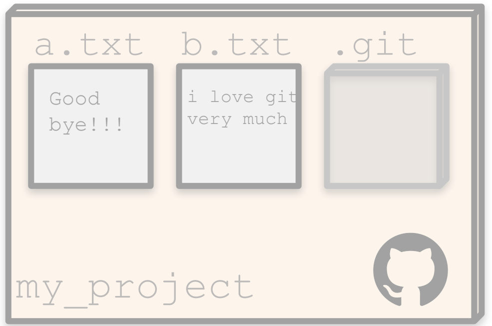
Using other's work
e.g. the practical works

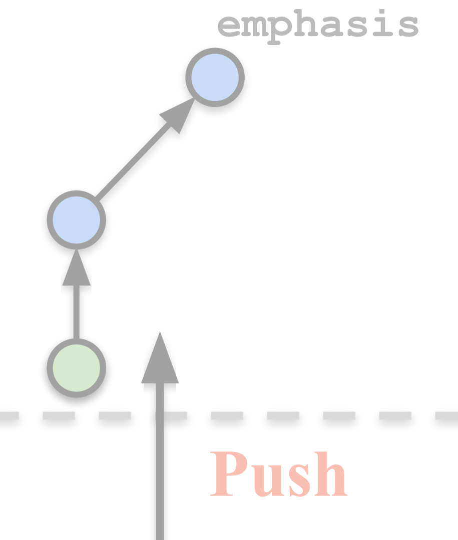
Working with others
e.g. your final project

↩Remote repos



Remote repo



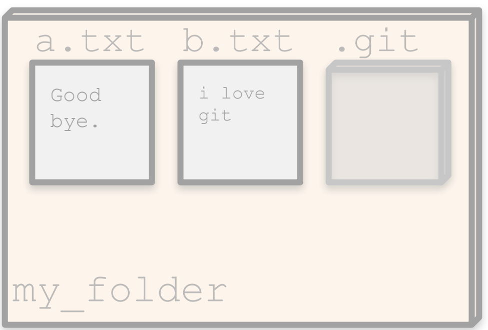


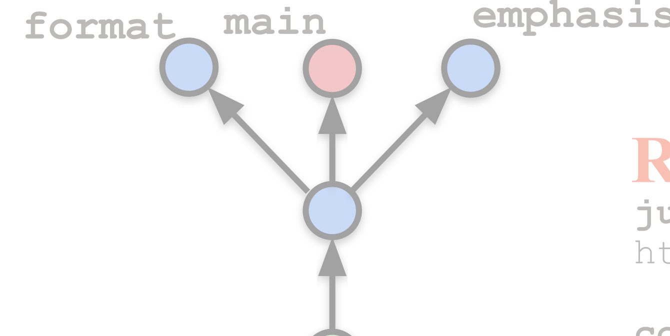
Remote

Local

Push

Repository (repo)



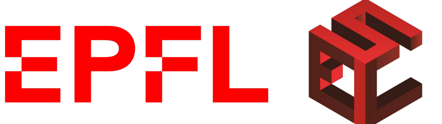


Remotes' list

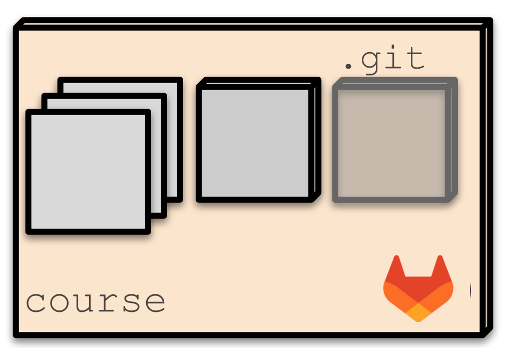
juan
https://github.com/juan/my_project.git

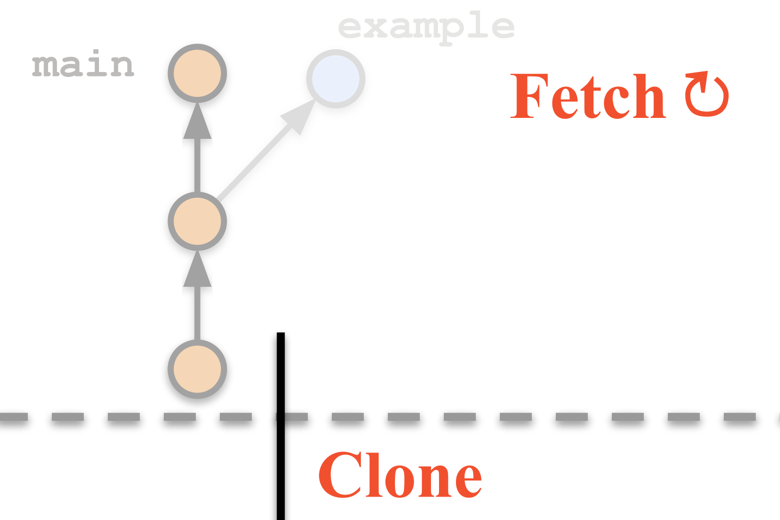
course
https://gitlab.epfl.com/juan/proj.git

Cloning a repo



Remote repo

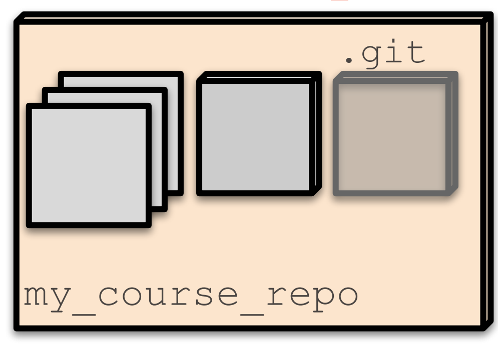


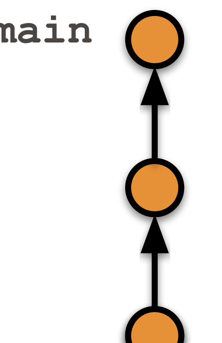


Fetch ↻

Clone

Local repo





Remotes' list

`course`
`https://gitlab.epfl.com/esl/course.git`

Fetch only refreshes information

Cloning a repo

Remote repo

.git

course

main

example

Pull

Local repo

.git

my_course_repo

main

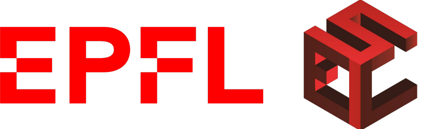
main

Remotes' list

course

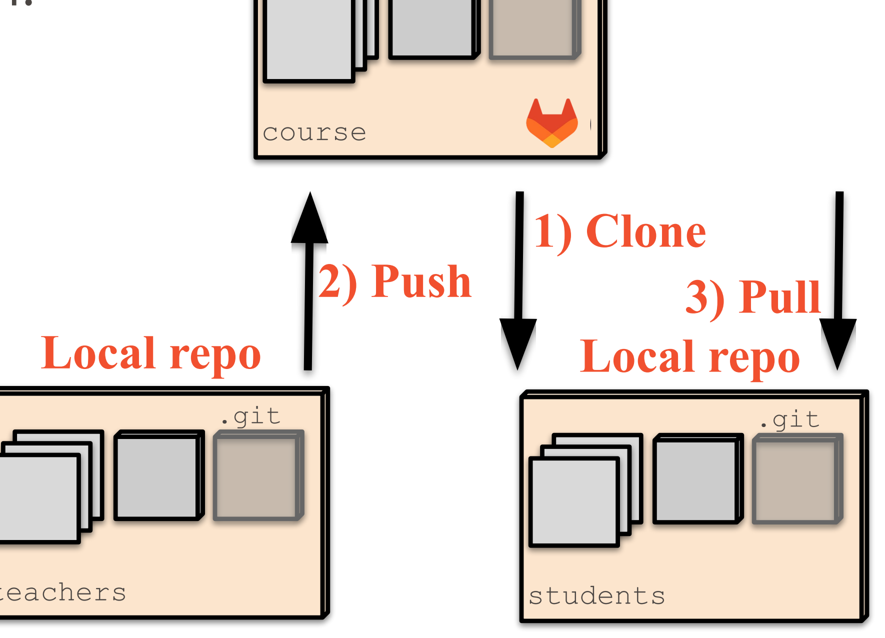
https://gitlab.epfl.com/esl/course.git

Demo

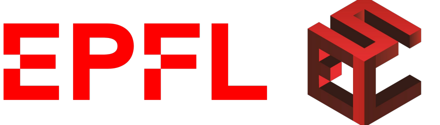


Clone a repo

- Rename the folder of the repo to “teacher”
- Clone the repo into “student”
- Add changes in the teacher’s version, push them.
- Pull those changes on the student’s version.

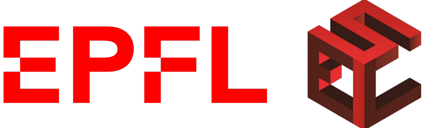


Demo - recap



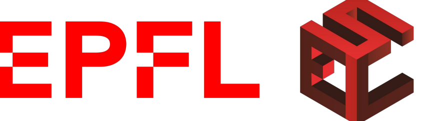
- `git clone <url> <folder>`
  - Clones the repo in <url> into <folder>
  - Adds the remote repo to the remotes' list
- `git remote rename <old name> <new name>`
  - Rename the link to the remote
- `git fetch --prune <remote name>`
  - Updates the information about the remote <remote>
  - **`--prune`** updates also deleted remote branches
- `git checkout -b <local branch> <remote>/<branch name>`
  - Creates a new local branch from a remote branch
- `git pull <remote> <remote branch>`
  - Pull changes from the remote branch

Demo - recap



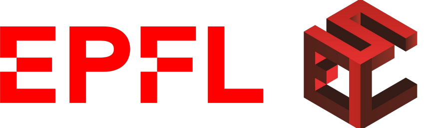
- `mv <source> <destination>`
  - **M**o**v**e (or rename) a file or folder

Demo - cheatsheet



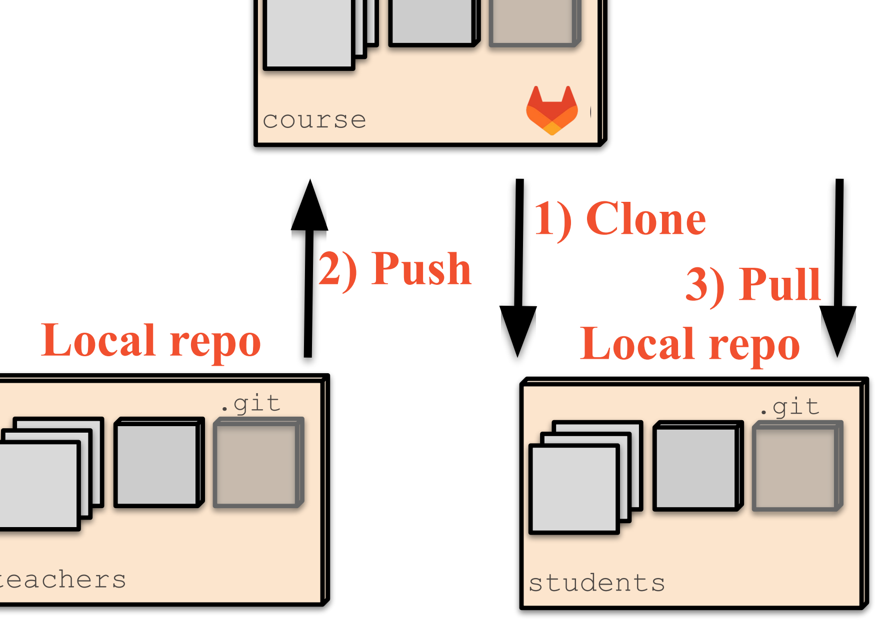
```
        # Rename the local version to "teacher"
ll                              # Check where we are
cd ..                           # Go one folder back
mv my_project teacher           # Rename my local repo
cd teacher                      # Go back into the repo
git remote                      # Check the name of the remote
git remote get-url course       # Get the url of the remotes in the list, assuming you called it "course"

        # Clone the remote repo into another folder called "student"
cd ..                           # Go back, you don't want to clone into the other repository
git clone <url> student         # Clone the remote version into the local
cd student                      # Get into the repository
# Note that the content of the two local repos is not the same, since not all the latest changes have been pushed.

        # Push some other brnaches and commits to the repo from the teacher's side
cd ../teacher
git push course --all

        # Get the changes
git remote                                  # Check the name of the remote
git remote rename <old_name> nds_course     # Rename it to something relevant
git branch -a                               # You should still not see the new branches in the remote
git fetch                                   # Update the info we have of the remote
git branch -a                               # You should now see the new branches
git status                                  # Check that we are outdated
git checkout -b main nds_course/main        # Get the new branch

        # Add new changes
cd ../teacher
# Make changes
git commit -am "Updated file"
git push main
cd ../student
git pull main                               # If it fails, git might not be aware of the fact that there were changes
git fetch main
git pull main
```

Demo - try it yourself

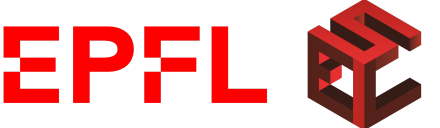


Clone a repo

- Rename the folder of the repo to “teacher”
- Clone the repo into “student”
- Add changes in the teacher’s version, push them.
- Pull those changes on the student’s version.



Handling the PWs



PW2
main
PW3
PW1
.git
personal
.git
course

Push

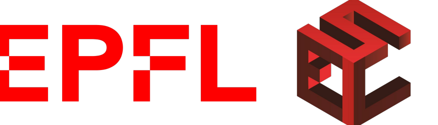
Pull

PW2
main
PW3
PW1
.git
my_course_repo

Remotes' list

course
https://gitlab.epfl.com/esl/course.git
juan
https://gitlab.epfl.com/juan/personal.git

Handling the PWs



PW2
b2
PW3
PW2
c1
b1
C
PW1
B
a1
A

personal

course

C
B
A

push juan PWx

pull course main

PW2
b2
PW3
PW2
c1
b1
C
PW1
B
a1
A

my_course_repo

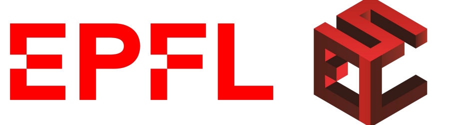
checkout -b PWx
commit

checkout main

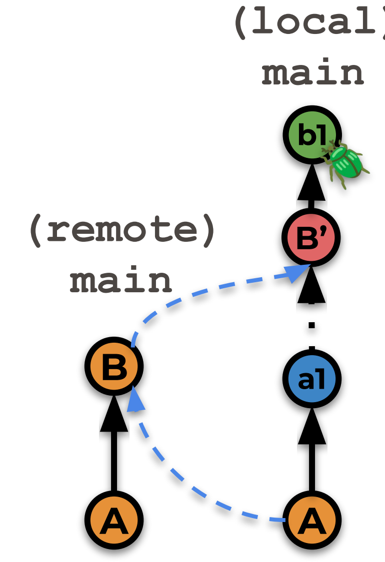
Remotes' list

course
https://gitlab.epfl.com/esl/course.git
juan
https://gitlab.epfl.com/juan/personal.git

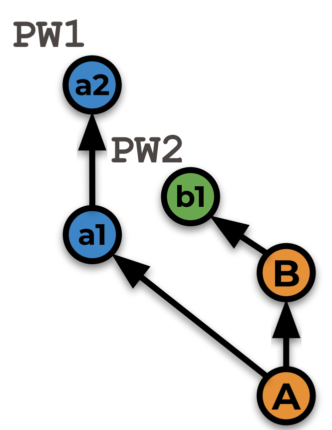
Handling the PWs



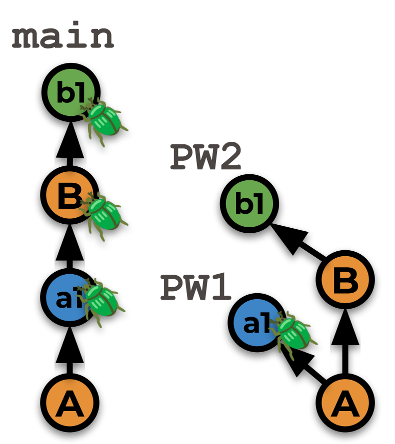
You don't have to merge main ever (pull)



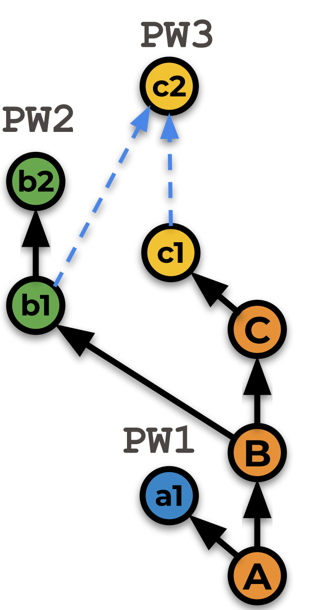
Your PW are independent (checkout)



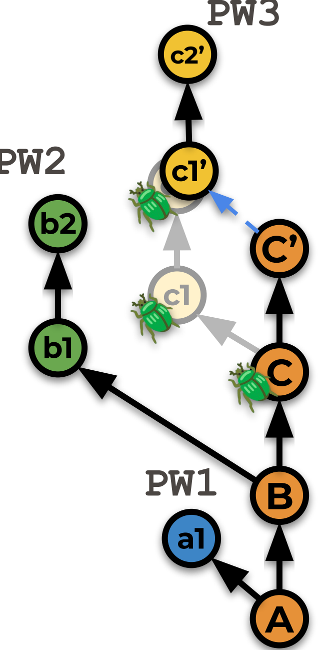
Avoid carrying around "rookie mistakes"



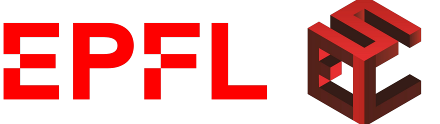
Re-use your solutions (merge)



Get bug-fixes (pull & rebase)



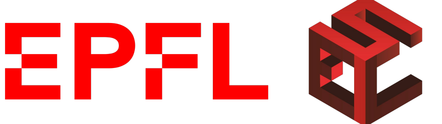
Demo



The practical works' dynamic

- Have two local repos: “teacher” and “student”
- Have a remote repo where “teacher” pushes, and “student” pulls
- Have a remote repo where the “student” pushes
- Do the PW flow
  - Add a commit with a new PW on the teacher side, and push
  - Pull it on the student side
  - Create a branch from there, and commit changes
  - Push those changes to the personal repo of the student
  - Checkout to main and repeat…

Tips



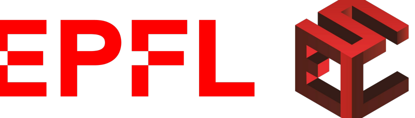
Keep your main branch clean!

- ✓ **Create branches** for PWs or tests
- ✓ **Checkout to main** before pulling from the course's repo
- ✓ **Commit changes** before checking out to main
  (or stash or restore them)

As a rule of thumb

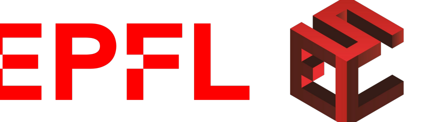
```
1. git checkout main
2. git pull course main
3. git checkout -b PWx
4. git add <files>
5. git commit -m "Some relevant message"
6. git push <personal remote> PWx
```

Demo - cheatsheet




```
        # Create a personal repo for the student
cd student                                    # Go into the "student" folder
git remote add personal <url>                 # Add the new repo to your list of remotes
git push student --all                # Push all you have to your personal backup
git remote rename <name of the other repo> nds_course    # Give them very explicit names so you don't get them mixed up

        # Add a new PW on the teacher's side
cd ../teacher
# Add a file PW1.txt
git add PW1.txt
git commit -m "Added PW1"
git push -u course main          # This assumes that your remote is called course, and your branch is called main. If you see problems, maybe go back to previous demos and see how we changed the name of the remote or the branch!

        # Pull those changes from the student side
cd ../student
git checkout main                # Make sure you are on the main branch
git pull nds_course main         # Pull all the changes from the course

        # Create a new branch to work on your PW
git checkout -b PW1
# Make changes to the PW
git commit -am "Solved the pw1"
git push personal PW1            # Push those changes to your personal back up

        # Add a new PW on the teacher's side
cd ../teacher
# Add a PW2.txt file
git add PW2.txt
git commit -m "Added PW2"
git push

        # Repeat the process on the student side
cd ../student
git checkout main
git pull nds_course main
git checkout -b PW2
# Solve the exercise
git commit -am "Solved pw2"
git push personal PW2

        # Fix a problem on PW1
git checkout PW1
# Fix whatever
git commit -am "Fixed an issue"
git push personal PW1
```

Demo - try it yourself



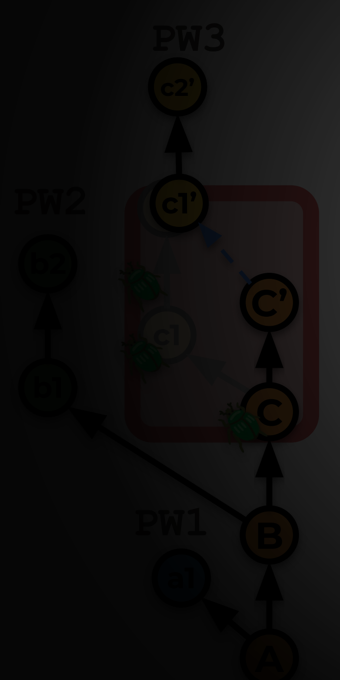
The practical works' dynamic

- Have two local repos: “teacher” and “student”
- Have a remote repo where “teacher” pushes, and “student” pulls
- Have a remote repo where the “student” pushes
- Do the PW flow
    - Add a commit with a new PW on the teacher side, and push
    - Pull it on the student side
    - Create a branch from there, and commit changes
    - Push those changes to the personal repo of the student
    - Checkout to main and repeat…

Conflicts

Re-use your solutions (merge)

Get bug-fixes (pull & rebase)



c1'

c1

I rarely HATE
learning
about git.

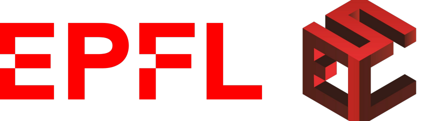
C'

I really enjoy
learning
about git.

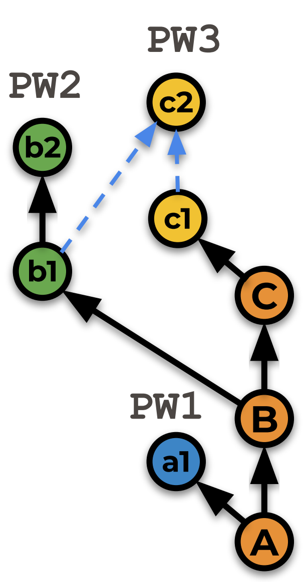
C

I rarely enjoy
learning
about git.

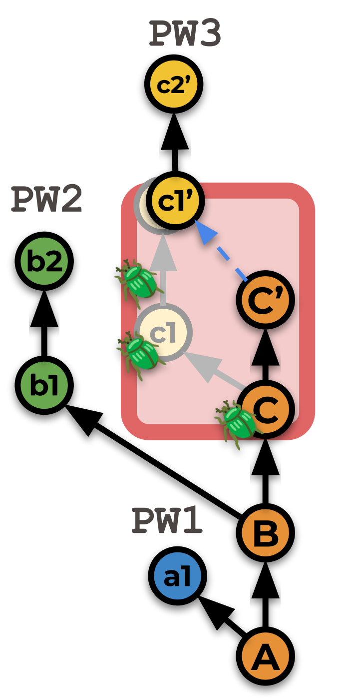
Conflicts

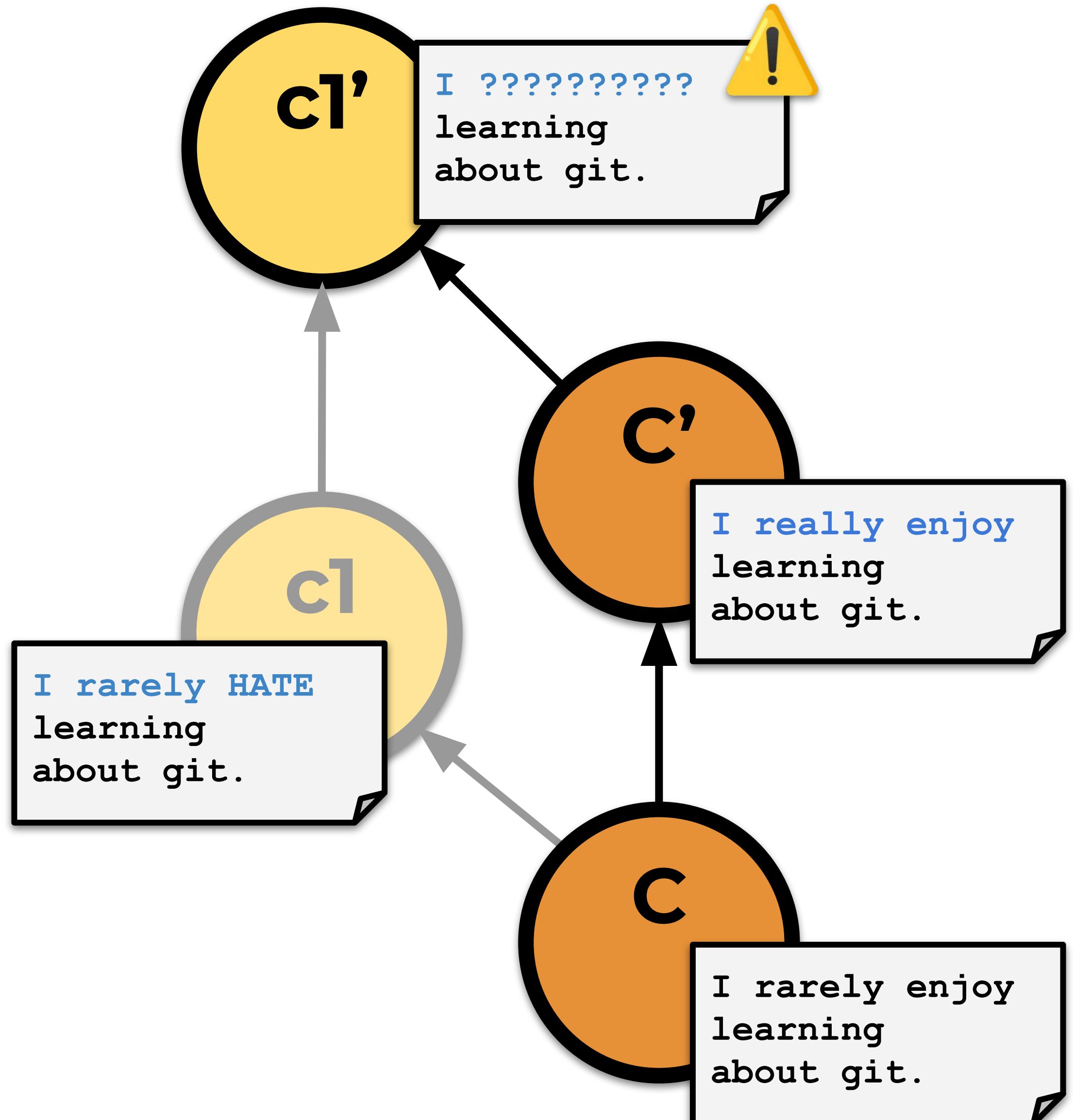


Re-use your solutions (merge)

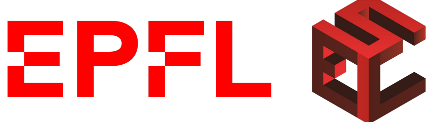


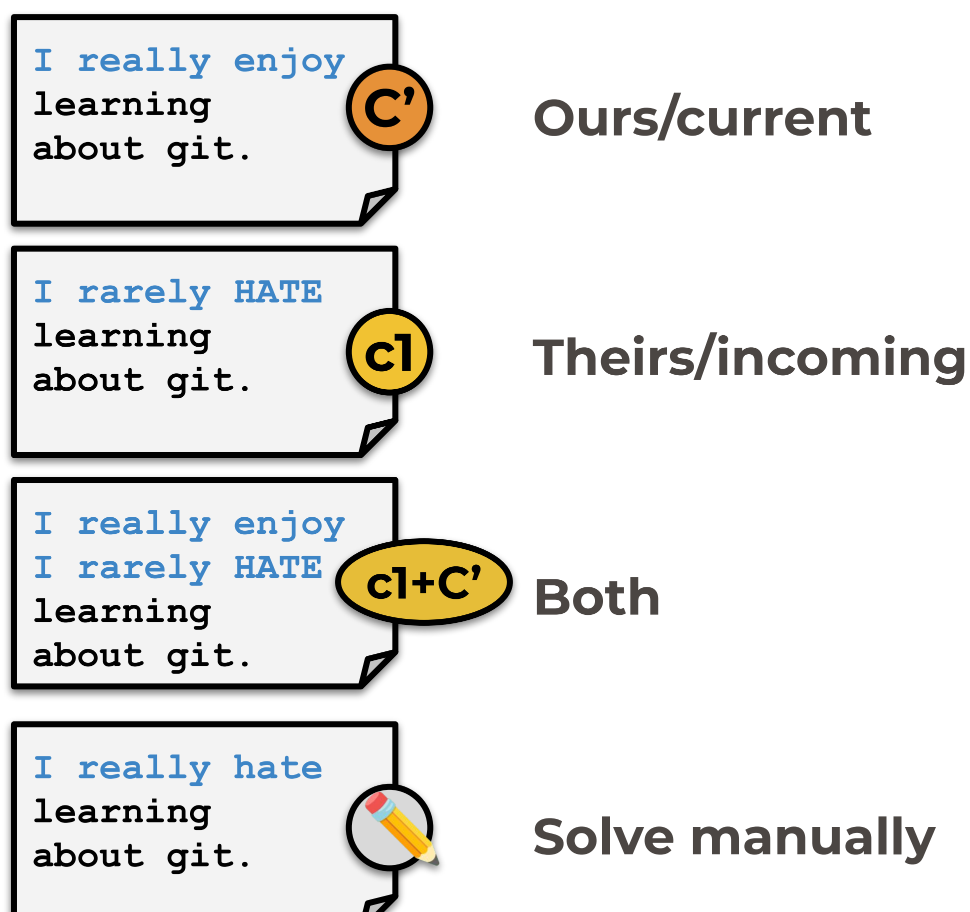
Get bug-fixes (pull & rebase)





Conflicts





Ours/current

Theirs/incoming

Both

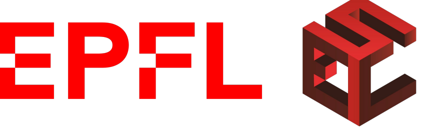
Solve manually

```
$ git rebase main
Auto-merging a.txt
CONFLICT (content): Merge conflict in <file>
error: could not apply fec7715... c1
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git
rebase --abort".
Could not apply fec7715... c1
$ git checkout --ours <file>
$ git add <file>
$ git rebase --continue
```

```
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
<<<<<<< HEAD (Current Change)
i really enjoy learning git
=======
i rarely HATE learning git
>>>>>>> fec7715 (corrected logic of message) (Incoming Change)
```

Solve **conflicts** in VScode: be well aware of what you are doing

Demo

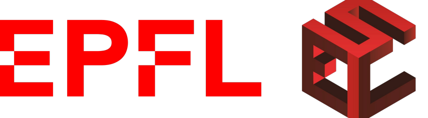


	Original (main)	Commit (main)	Commits (feature)		Rebased (feature)
a.txt	Original first line Original second line	@Change that will NOT be kept Original second line	#Change that will be kept Original second line	terminal →	#Change that will be kept Original second line
b.txt	first second third fourth fifth sixth seventh eighth	first @SECOND third fourth fifth sixth @7th eighth	first #2nd third fourth fifth sixth #SEVENTH eighth	vscode →	first @SECOND third fourth sixth seventh @SEVENTH eighth
c.txt	Original first line		#This file will be discarded	terminal →	
d.txt	Original first line Original second line	@This change will be kept Original second line	#This change will NOT be kept #This change will be skipped altogether Original second line	terminal →	@This change will be kept Original second line

Demo - recap

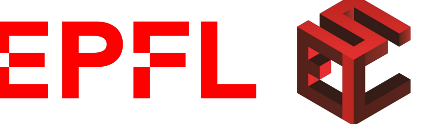
- `git checkout --ours/theirs <file>`
  - to choose which version to keep
  - **Use VSCode** (or any graphic user interface) to see what you are solving
- `git rebase --continue/skip`
  - Apply the next commit

Demo - cheatsheet



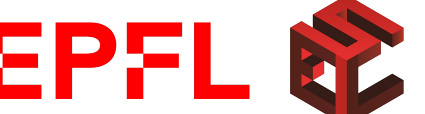

```
        # Baseline
# Make sure you have a, b, c, and d.txt on main. On a and b.txt, have two lines of text divided by an empty line
git commit -am “Baseline commit”

        # Add conflicting changes on a feature branch
git checkout -b myfeature
# Modify in a.txt - first line - “#Change that will be kept”
# Modify in a.txt - last line - “#This change will not cause any conflicts”
# Modify in b.txt - first line - “#Change that will be kept”
# Modify in b.txt - last line - “#Change that will be fused manually”
# Modify in c.txt - first line - “#This file will be discarded”
# Modify in d.txt - first line - “#This change will NOT be kept”
git commit -am “Made several changes to cause conflicts”
# Modify in d.txt - first line - “#This change will be skipped altogether”
git commit -am “made an additional change”

        # Add conflicting changes on the main branch
git checkout main
# Modify in a.txt - first line - “@Change that will NOT be kept”
# Modify in a.txt - third line - “@This change will be kept despite selecting --theirs, because it’s not conflicting”
# Modify in b.txt - first line - “@Change that will NOT be kept”
# Modify in b.txt - last line - “@Change that will also need to be fused manually”
# Delete c.txt
# Modify in d.txt - first line - “@This change will be kept”
git commit -am “Commit on main that will definitely cause conflicts with my feature”

        # Rebase and solve conflicts
git checkout myfeature
git rebase main
git checkout --theirs a.txt
git add a.txt

# On VSCode, accept the incoming change on the first line
# On VSCode manually fuse the changes to the last line
git add b.txt

git rm c.txt

git checkout --ours d.txt
git add d.txt

git rebase --continue
git am --show-current-patch
git rebase --skip
```

Demo - try it yourself



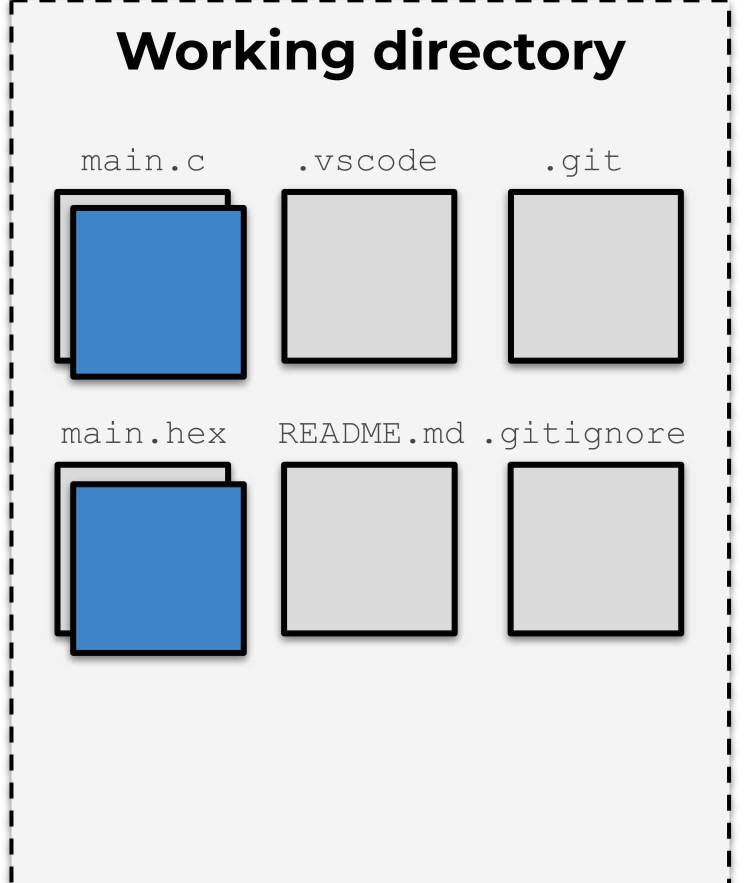
	Original (main)	Commit (main)	Commits (feature)		Rebased (feature)
a.txt	Original first line Original second line	@Change that will NOT be kept Original second line	#Change that will be kept Original second line	terminal →	#Change that will be kept Original second line
b.txt	first second third fourth fifth sixth seventh eighth	first @SECOND third fourth fifth sixth @7th eighth	first #2nd third fourth fifth sixth #SEVENTH eighth	vscode →	first @SECOND third fourth sixth seventh @SEVENTH eighth
c.txt	Original first line		#This file will be discarded	terminal →	
d.txt	Original first line Original second line	@This change will be kept Original second line	#This change will NOT be kept #This change will be skipped altogether Original second line	terminal →	@This change will be kept Original second line

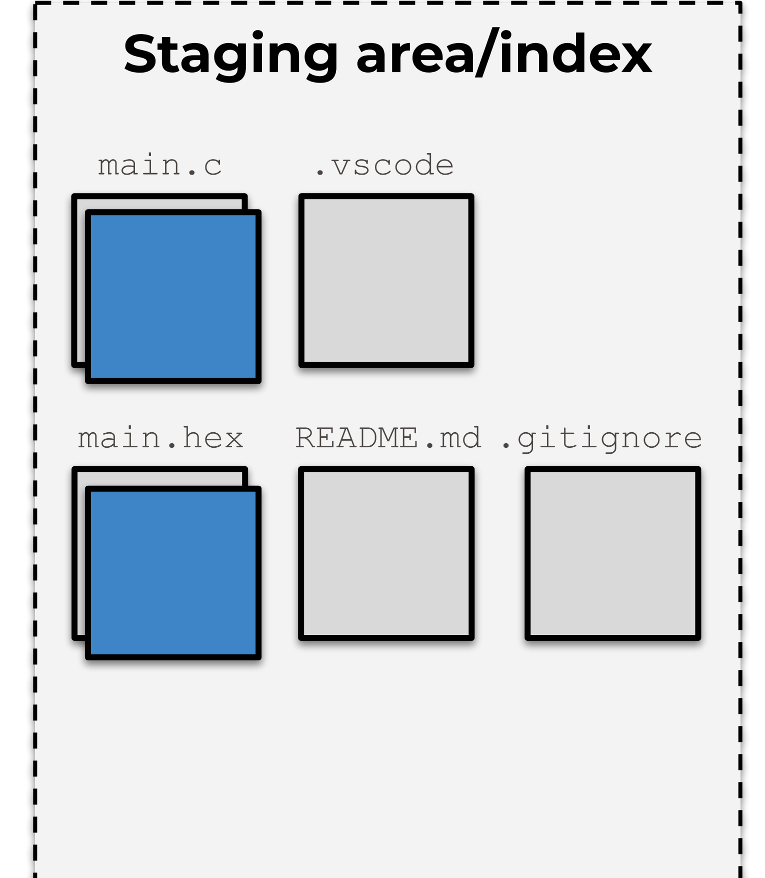
Ignoring files

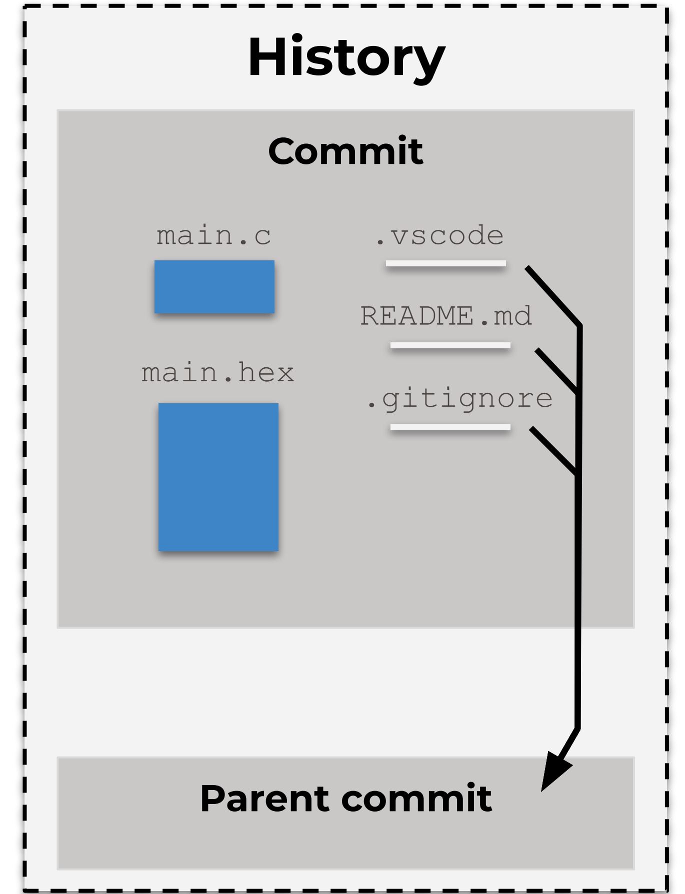


git add

git commit







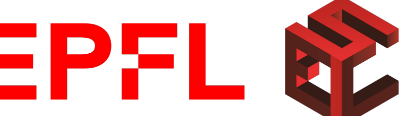
Ignoring files

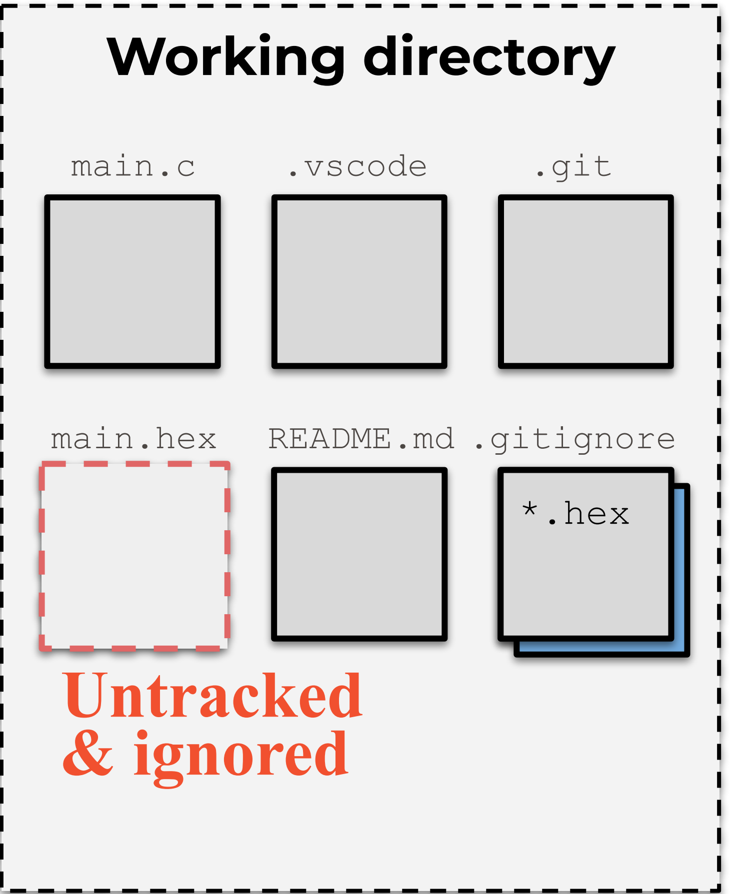
.gitignore

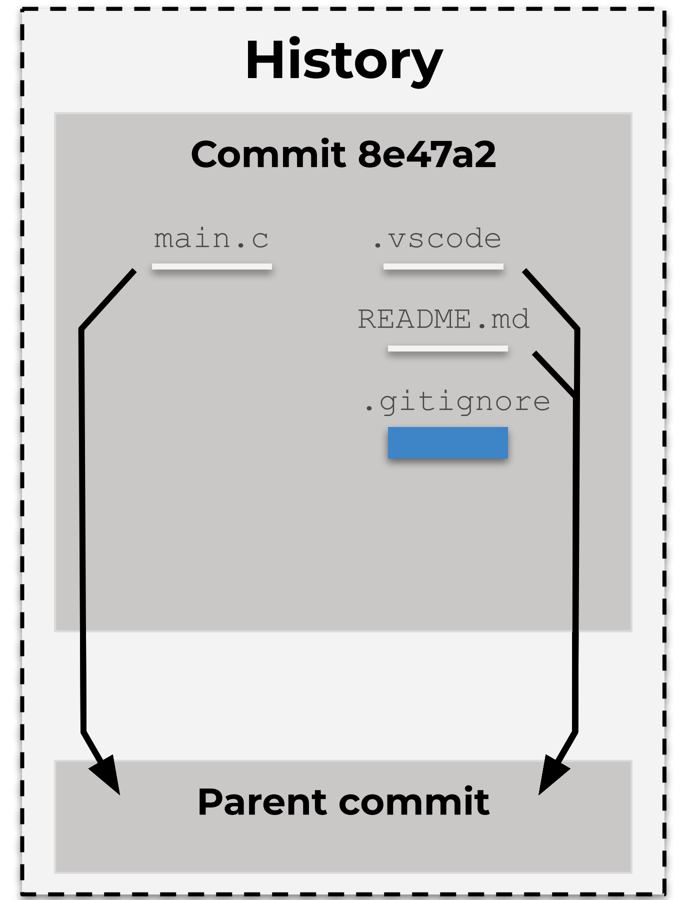
```
# One particular file
build/main.hex

# A type of file
*.hex

# Whole folders
build/
```

Use the **.gitignore** wisely: make sure you track important files

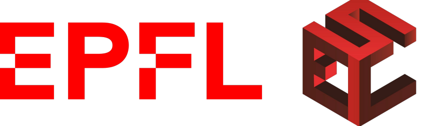
Ignoring files

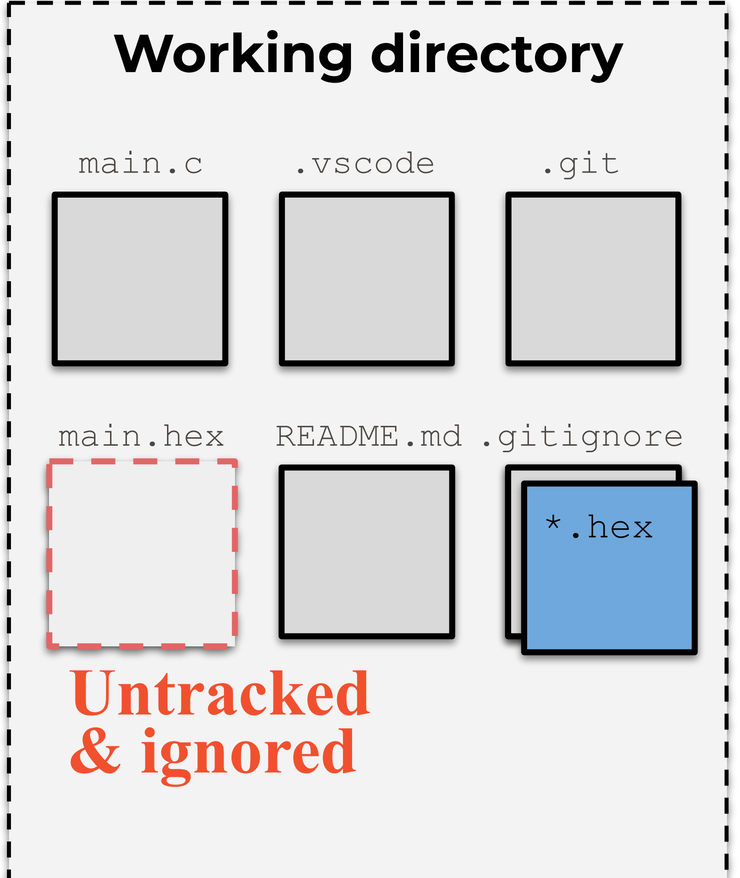


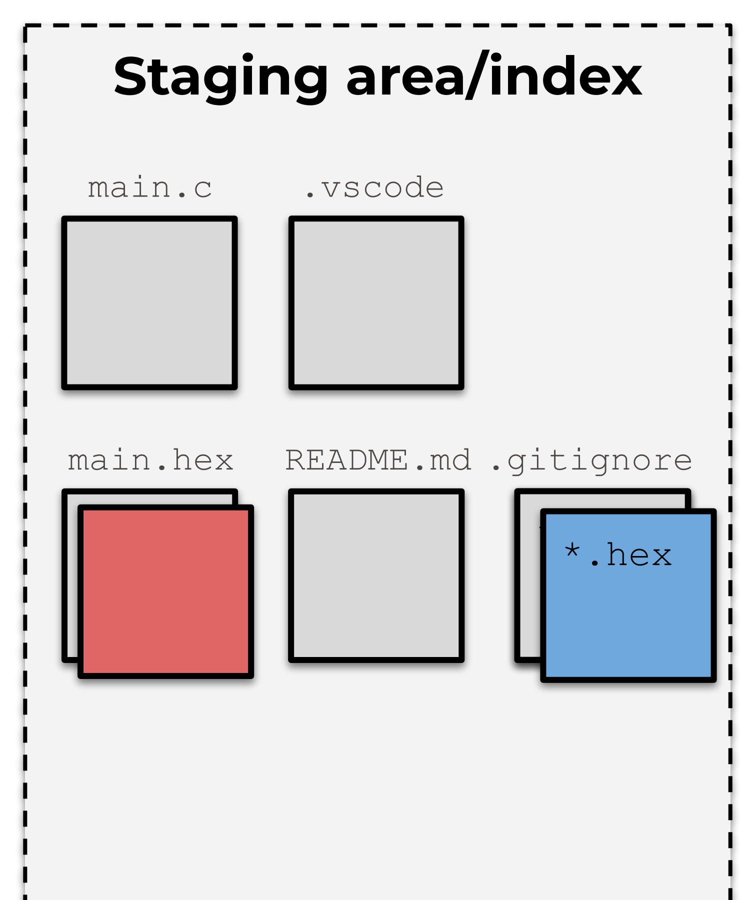




Demo



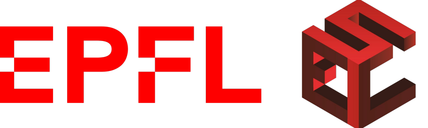




Ignoring files

- Untrack a file
- Add it to the gitignore
- Commit those changes
- Test it!

Demo - tips

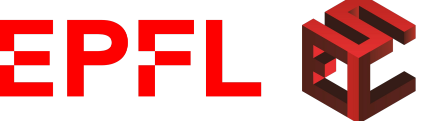


- 💡 `.gitignore` templates can be found in: **github.com/github/gitignore**

- 💡 Use git `git status -u` to show **u**ntracked files and make sure you are not leaving any relevant file behind

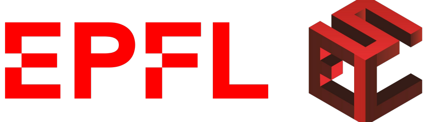
- 💡 `.gitignore` is only a gatekeeper/bouncer: will not remove files if they are already tracked

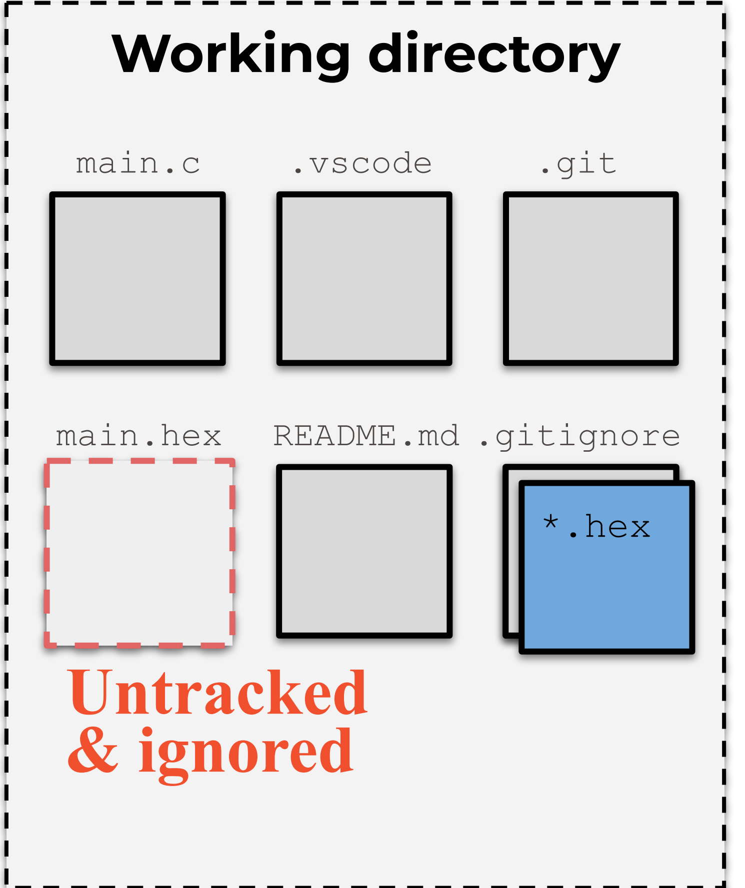
Demo - cheatsheet

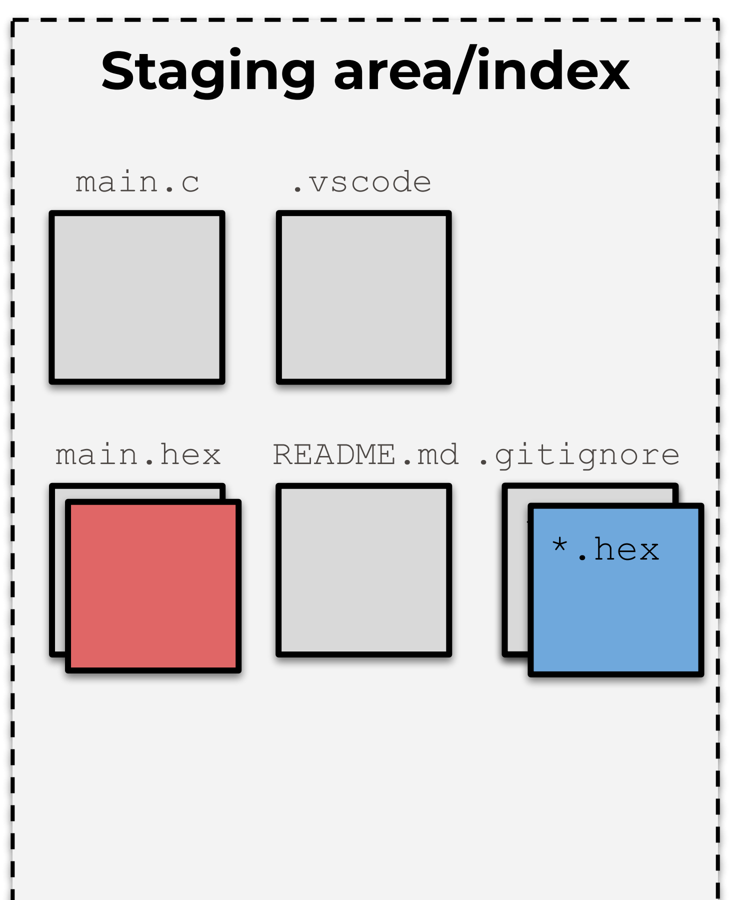



```
git rm --cached a.txt                             # Select a file and remove it from the staging area
git commit -m “Removed a.txt from staging area”   # Commit
git status -u                                     # See untracked files that are not being ignored
# Create a .gitignore file and add “a.txt”
git status -u                                     # See that a.txt is no longer listed
git add .gitignore
git commit -m “Added a gitignore with a.txt”
# Make some modifications to a.txt
git status                                        # Notice how they don’t show up in git status
# Modify .gitignore to say “*.txt” instead (ignore al .txt files)
git status                                        # We will see modification in the gitignore file, but not on a.txt
# Make modification to b.txt
git status                                        # Notice that modifications to b.txt ARE shown in git status, because b.txt was already in the staging area!
```

Demo - try it yourself







Ignoring files

- Untrack a file
- Add it to the gitignore
- Commit those changes
- Test it!

EPFL
Thank you!
EPFL - Embedded Systems Laboratory