CS6 Practical System Skills

Fall 2019 edition

Leonhard Spiegelberg
lspiegel@cs.brown.edu

14 Git

CS6 Practical System Skills

Fall 2019

Leonhard Spiegelberg *lspiegel@cs.brown.edu*

14.00 Git resources

Official git book: https://git-scm.com/book/en/v2

Atlassian tutorial: https://www.atlassian.com/git/tutorials
(many slides are actually based on this)

Learn Version Control with Git by Tobias Günther. ISBN: 9781520786506
available freely here: https://www.git-tower.com/learn/git/ebook/en/command-line/introduction

Cheatsheets:
https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf
https://about.gitlab.com/images/press/git-cheat-sheet.pdf
https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet

⇒ Best however is learning by doing!

14.01 Why?

⇒ We have all been there…

- Code worked, then we made changes and it broke…
  → How to revert to an old version?
- Taking snapshots / versions in folders for backups
  → e.g. version01, version02, …?
  → keeping a changelog.txt?
- Using a Dropbox folder or gdrive to share a project?
  → let's have only one person working on the document to avoid conflicts or breaking changes…?

14.01 Version control

Version control system: software that tracks and manages changes on a set of files and resources

→ systems usually designed for software engineering projects

Version control systems typically enable

⇒ automated versioning

⇒ to help teams to collaborate on projects

⇒ to automatically backup files

⇒ efficient distribution of updates via deltas

14.01 What files should be put under version control?

⇒ **definitely:** text files

- source files
- build files / make files

⇒ **depends:** project files and small binary files

- ok, if they do not change. E.g. small image files, ...

⇒ **strictly no:** large files, temporary files, **!!!PASSWORDS!!!**

14.02 Version control

⇒ There are multiple version control systems, popular are

git (this the defacto standard)

subversion

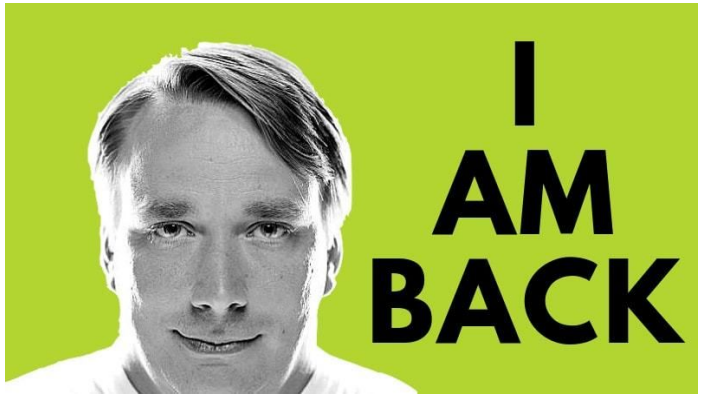
mercurial

\+ many more that are not used anymore or are commercial

14.03 Git

Created by Linus Torvalds in 2005
⇒ designed for linux kernel development

→ https://www.youtube.com/watch?v=4XpnKHJAok8



Git design goals:
⇒ speed
⇒ support for non-linear development and thousands of parallel branches
⇒ fully distributed
⇒ able to handle large projects efficiently

⇒ Git is the standard VCS used today.

14.03 Installing git

Mac OS X:

```
brew install git (brew.sh)
```

git is already installed on department machines

Debian/Ubuntu:

```
sudo apt-get install git
```

You can install also git under windows, but it's most stable under *NIX systems.

Fedora/Redhat:

```
sudo yum install git
```

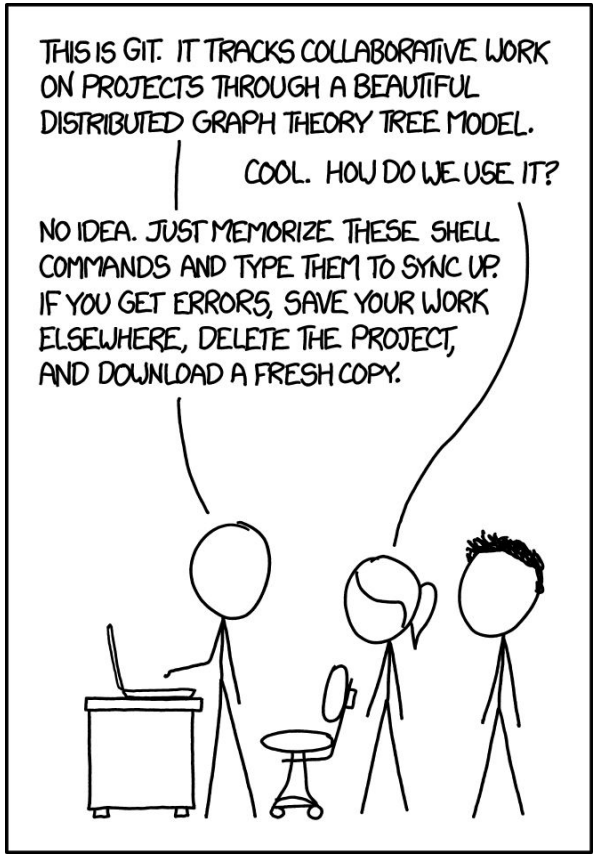
We use git version 2.xx.x

14.04 How to use git?

Basic syntax:

```
git cmd parameters
```

⇒ to learn more about `cmd`, type `git help cmd`



14.04 Setting up git

Before running commands, we need to specify a user and email address under which changes are tracked.

```
git config --global user.name "tux"
git config --global user.email "tux@cs6.brown.edu"
```

⇒ use `git config --list` to list config params

14.05 Repositories

Repository = a virtual storage of your project.

⇒ Make a folder a repository by running `git init` within it.

→ this will create a (hidden) folder `.git` where all versioning files are stored.

Alternative: Run `git init project_directory`

⇒ If a folder is a git repository and on your machine, it's referred to as local repository too.

14.05 Using a specific user for a local repository

You can use a repo-specific user for all your changes, to do so

⇒ instead of `git config --global`, use `git config user.name "tux"` to set repo-specific user name

⇒ Analog, `git config user.email "tux@cs.brown.edu"`

Note: You can also use git config --local, which is the same as git config.

14.05 Git basics

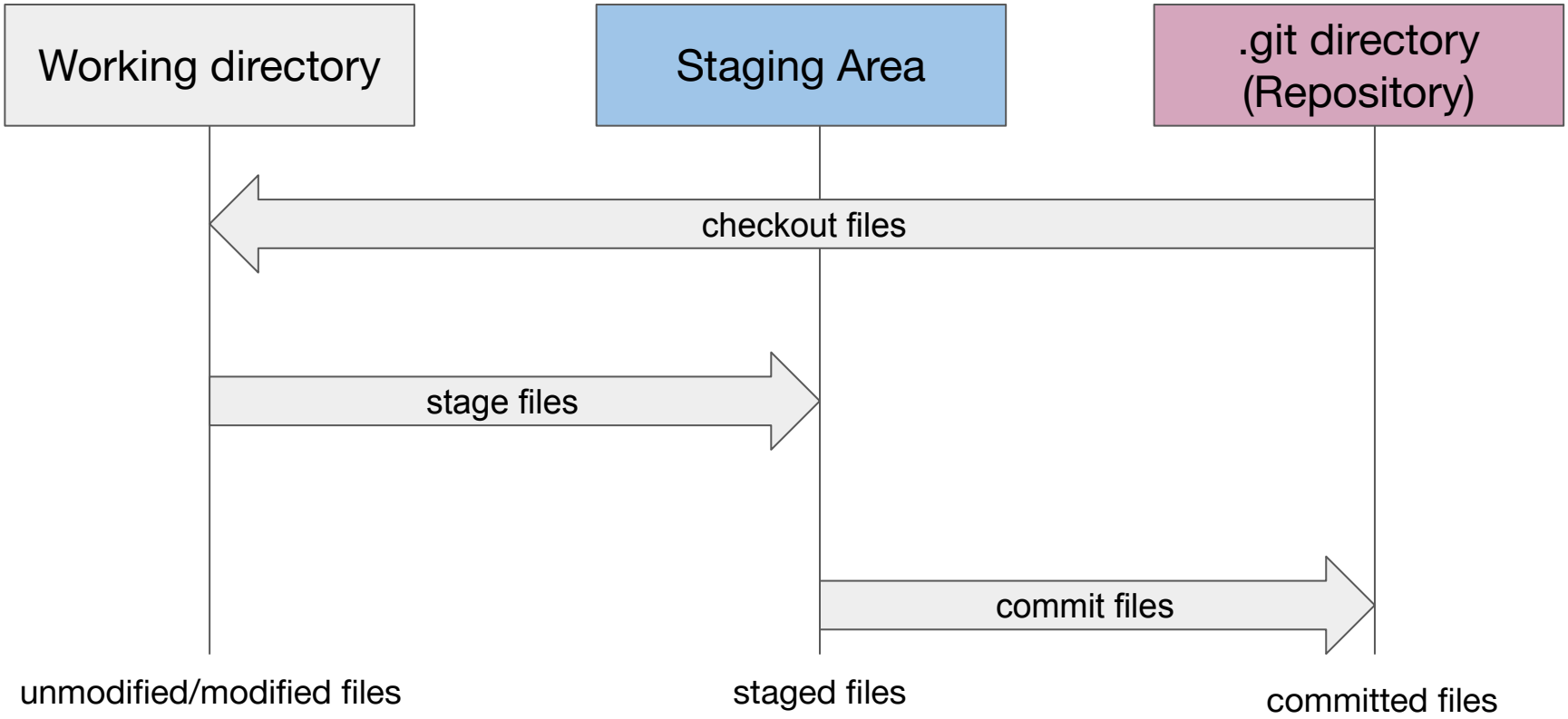
⇒ When working in git, you edit your local files in your local working copy (aka local repository)

⇒ When you're done with your edits, you bundle changes in a **commit** which can be seen as a snapshot.

⇒ git has three areas for a local git project/repository:

1. working area (i.e. working directory)
2. staging area
3. git area (i.e. git directory/repository)

14.05 Git areas



14.05 Basic workflow - creating a commit

(1) Modify files in your working directory

(2) stage files by appending to the staging area (`git add`)

(3) commit files, which takes all files in the staging area and creates a snapshot to be permanently stored as changeset in the `.git` directory (`git commit`)

14.05 Staging files - git add

⇒ use `git add` to add files to the staging area

⇒ moves a file from untracked status to tracked status if it was previously not tracked.

Syntax:

```
git add file

git add directory        # e.g. git add .

git add *.html           # using wildcard pattern
```

14.05 Checking which files are staged

⇒ Use `git status` to get an overview of files

→ per default, long output enabled. Use `-s` for shorter output

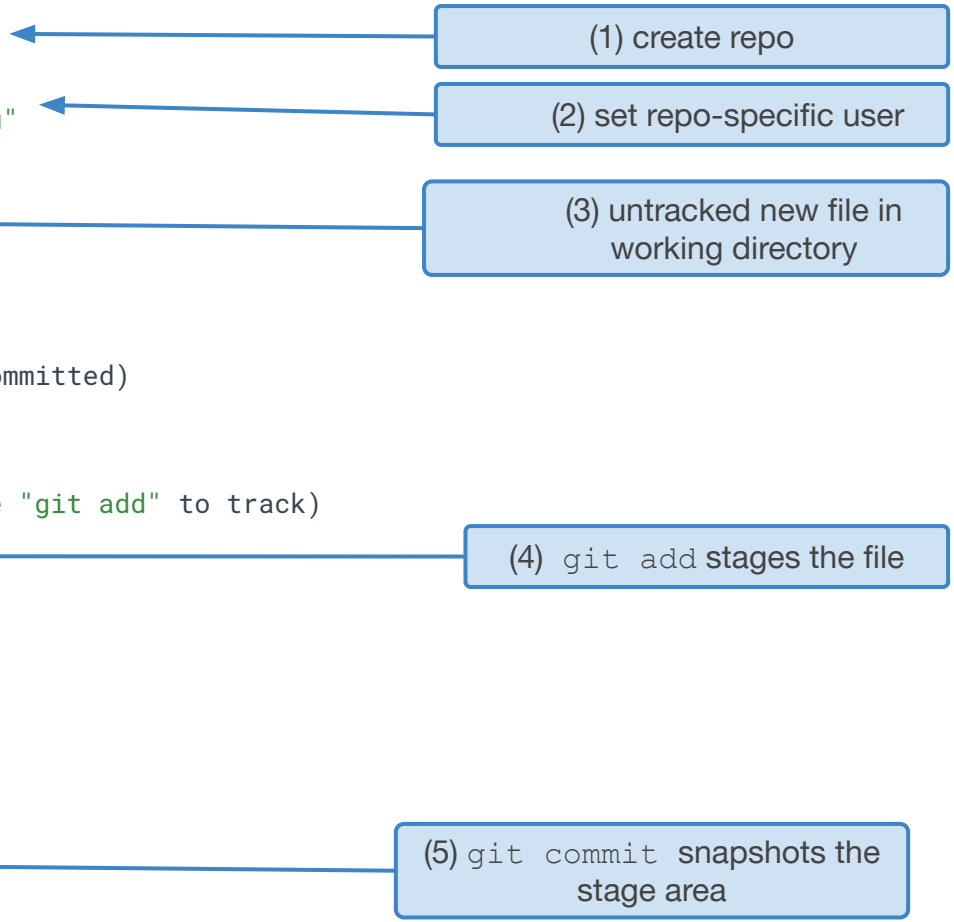
⇒ after you added a file, you can print the changes via

`git status -v` or `git status -vv`

⇒ After you staged all your files, you can create a commit via

`git commit -m "commit message"`

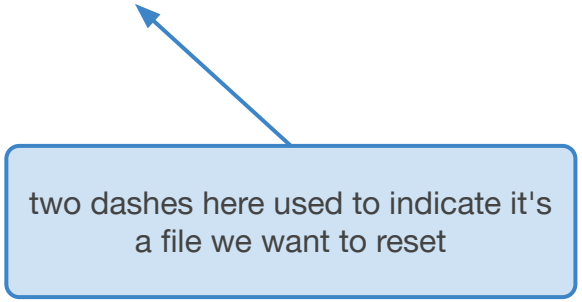
```
tux@cs6demo:~$  mkdir repo && cd repo && git init
Initialized empty Git repository in /home/tux/repo/.git/
tux@cs6demo:~$  git config user.name "tux"
tux@cs6demo:~$  git config user.email "tux@cs6server.edu"
tux@cs6demo:~$  echo "Hello world" > README.md
tux@cs6demo:~$  git status
On branch master

No commits yet

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

    README.md

nothing added to commit but untracked files present (use "git add" to track)
tux@cs6demo:~$  git add README.md
tux@cs6demo:~$  git status
On branch master

No commits yet

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

    new file:   README.md
tux@cs6demo:~$  git commit -m "initial commit"
[master (root-commit) 486a8b1] initial commit
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
tux@cs6demo:~$  git status
On branch master
nothing to commit, working tree clean
```



14.05 Unstaging files

To remove a file from the staging area, use

```
git reset -- file
```



Tip: you can set an alias for a command in git, i.e. to have `git unstage file` instead of `git reset -- file`, use `git config --global alias.unstage 'reset --'`

14.05 Removing files

⇒ Besides adding files, removal is tracked too.

⇒ There are two options to remove a file:

1. `git rm file`
2. `rm file && git add file`

⇒ git rm is basically a shortcut for rm && git add

⇒ **Note:** Rather than "removing" the file from version control, this creates a change which removes the file.

14.05 .gitignore

⇒ Sometimes you have files in your directory, which you don't want to track

→ can be folders, system files (e.g. on Mac OS X: .DS_Store)

⇒ create in dir of repository a file `.gitignore` with a glob pattern on each line to ignore certain files in this directory

⇒ you can also create a global .gitignore file, which is applied in addition to your local .gitignore

→ `git config --global core.excludesFile ~/.gitignore`

14.05 .gitignore example

.gitignore

```
# comments in ignore files are done via #
*.csv
ignored_folder/
```

/ indicates this is a directory to be ignored. This would NOT ignore a file called `ignored_folder`

Note: To add a file which is ignored by a .gitignore file, you can force add it via `git add -f file`

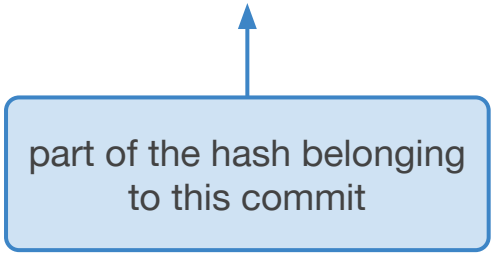
⇒ for more examples on gitignore files confer e.g.
https://www.atlassian.com/git/tutorials/saving-changes/gitignore

14.06 Commits

⇒ git creates for each commit a SHA-1 hash (40 character string of hex digits).

⇒ Often only part of the hash is shown, part of it or the full hash can be used to reference a specific commit.

```
tux@cs6demo:~$  git commit -m "initial commit"
[master (root-commit) 486a8b1] initial commit
```



14.06 Amending a commit

You can change a commit using `git commit --amend`

⇒ this opens up the editor you configured to be used by git (typically vim).

⇒ only use this on the last commit

⇒ can be used to change the commit message
and add or remove files.

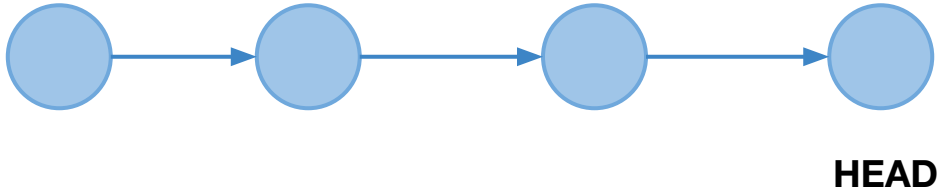
14.06 Undoing a commit

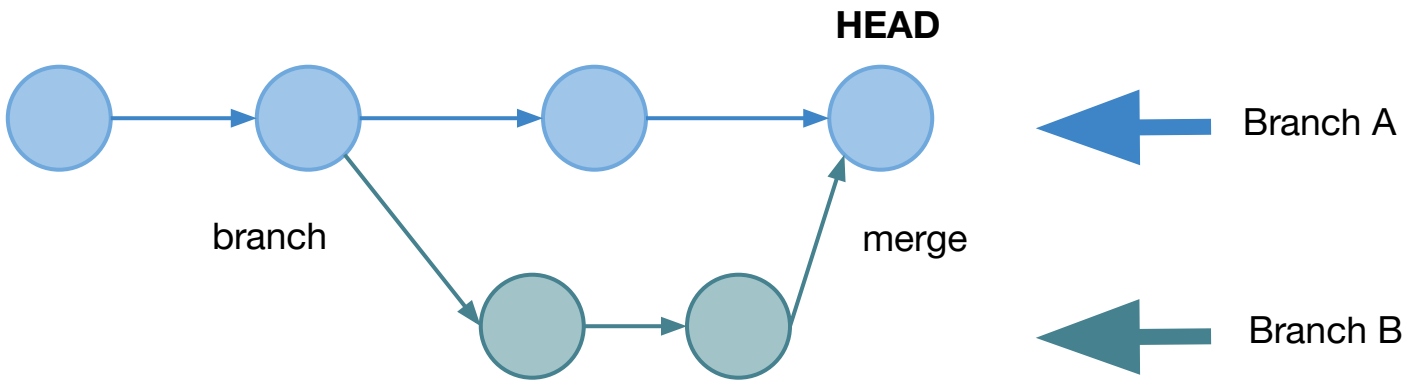
To undo a commit there are two options:
⇒ use `git revert`. Only use git reset if you know what you're doing.

1) `git revert [commit sha]` ⇒ creates a new commit which reverts changes
2) `git reset [--hard] commit` ⇒ reset repo to commit, if hard is specified discard changes.

14.06 Commits form a DAG

⇒ commits form a directed acyclic graph(DAG) with branches,
i.e. each node is a commit. Often (incorrectly) referred to as commit tree.



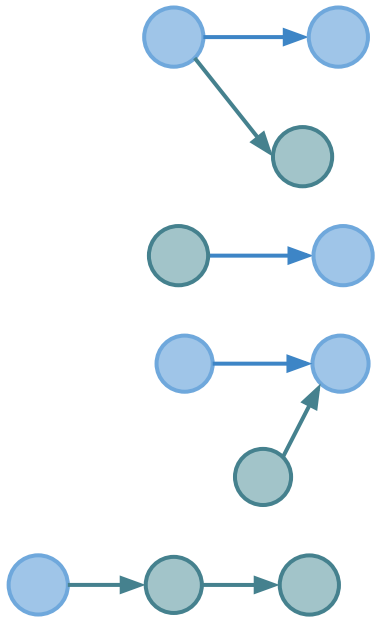


Branching

14.06 Manipulating the DAG

To work with the DAG, there are multiple commands:

git branch	branch out, i.e. create a new branch. Visually it is "forking" the DAG.
git checkout	going to a commit node
git merge	merging two commits together
git rebase	placing commits on top of each other



14.07 Creating a new branch

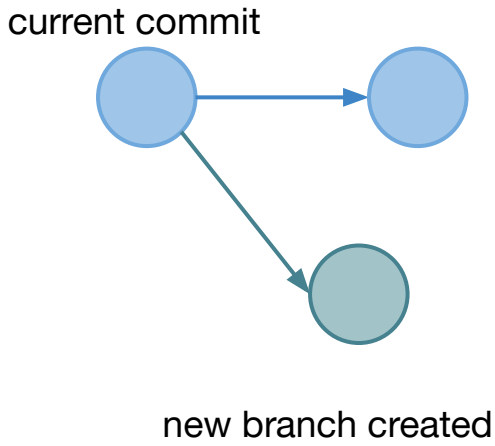
⇒ You can create a new branch via

```
git branch branch_name
```

or

```
git checkout -b branch_name
```

⇒ list existing branches via `git branch`



⇒ you can delete a branch via `git branch -d branch_name`

14.07 Checking out a branch or commit

⇒ you can checkout a branch or commit, i.e. replace the files in your working directory with the ones belonging to this commit

⇒ `git checkout hash` to go to a specific head

→ this will create a new, temporary branch called a detached HEAD

⇒ to checkout a branch (i.e. the tip or most recent commit of a branch), use `git checkout branch`

⇒ Example: `git checkout master`

⇒ Reminder: overview of branches via `git branch`

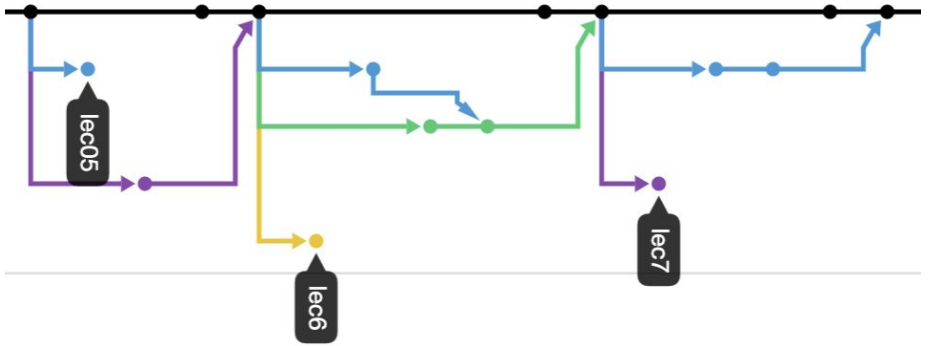
14.07 Checking out commits

⇒ use `git log` to see a history of commits with their messages

→ `git log -n` to print out n last messages.

→ `git log` has many more useful options to search through history

Website repo for course website:

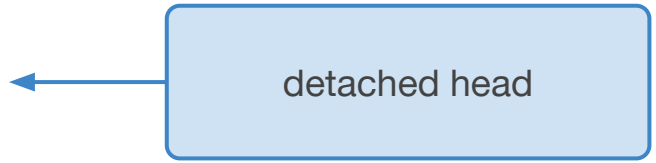


14.07 Checking out an old commit

⇒ sometimes you want to see an old version of your work, you can do so by checking out a commit via its hash (or parts of it)

⇒ Example: `git checkout f21a12b7abc`

```
tux@cs6demo ~$ git branch
* (HEAD detached at f21a12b)
  dev1
  master
```

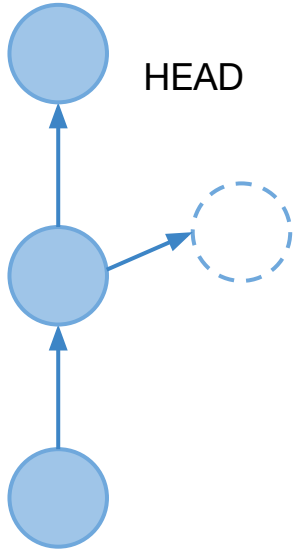


14.07 Checking out an old commit

⇒ You can also checkout a commit relative to the most recent one, for this do `git checkout HEAD~n`

⇒ Example: `git checkout HEAD~1`

⇒ you can create a new branch from a detached head via branch/checkout



Note: git checkout HEAD~0 creates a detached HEAD from the current head! git checkout HEAD just stays on the current commit

Bringing two branches together...
... there is only one issue:


How to resolve conflicts?

14.08 Joining two branches/commits

⇒ to join two branches (or commits), git provides two mechanisms:

1. `git merge` (today)
2. `git rebase` (next lecture)

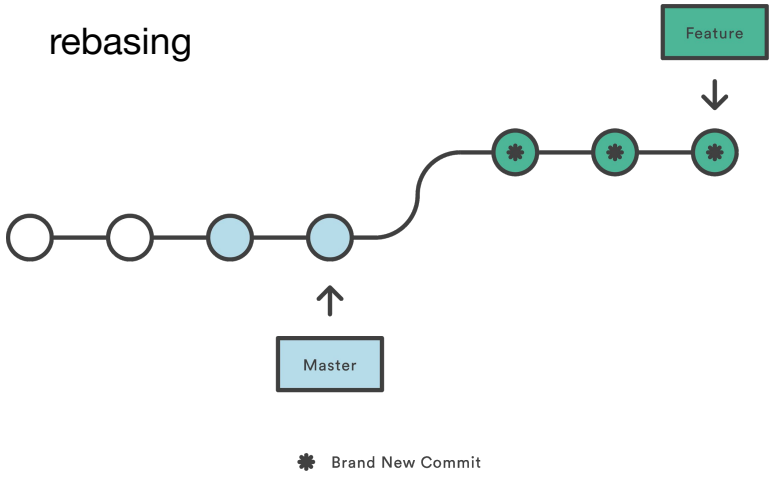
⇒ For now assume we want to join a branch feature with a branch master

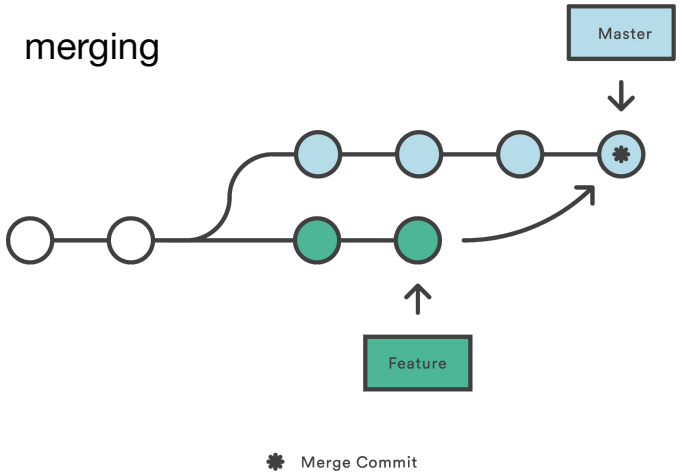
14.08 Merging vs. Rebasing

⇒ Rebase "replays" changes on top of the branch to rebase on

⇒ Merge creates a new merge commit

⇒ More on differences/use cases next lecture

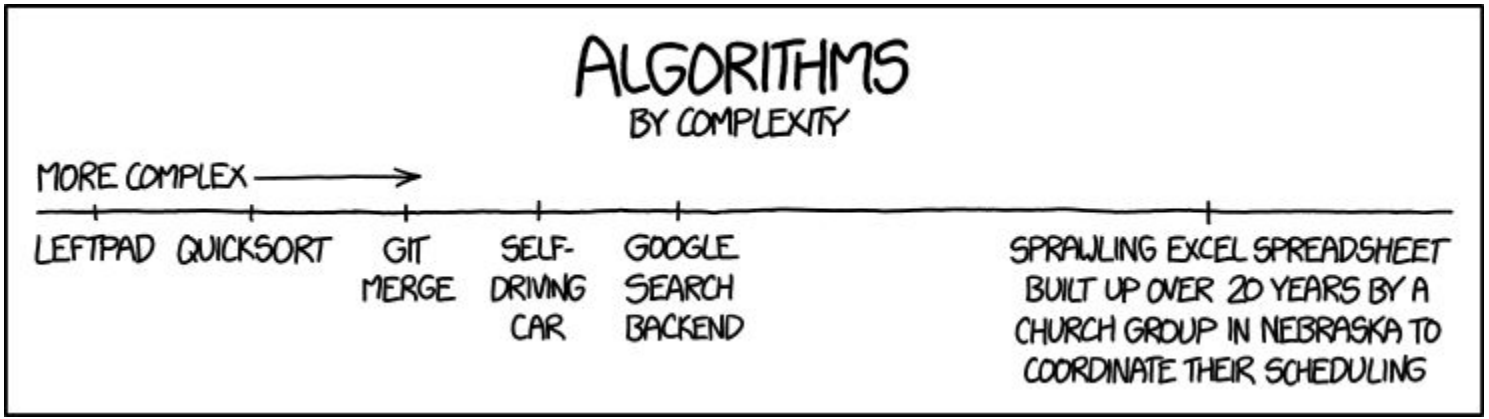




14.08 Merging two branches

git has an automatic merging algorithm, which often does a good job.

⇒ `git merge` can be configured to use different merge strategies



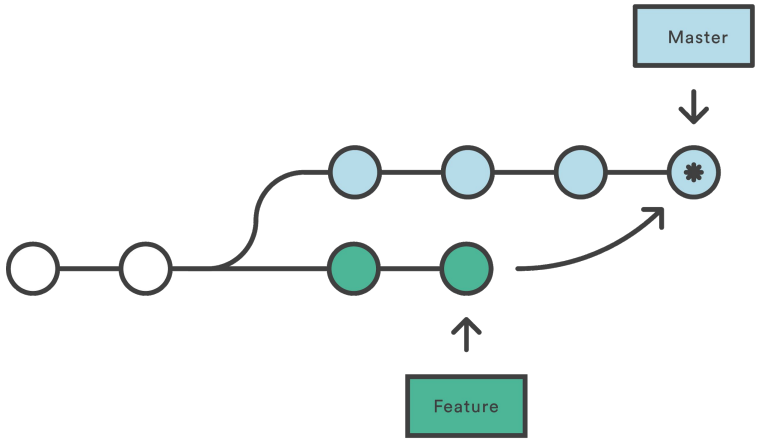
14.08 git merge

⇒ to merge a branch feature into a branch master do

```
git checkout master    # go to master branch
git merge feature      # merge feature into master
```

or as one-line version:

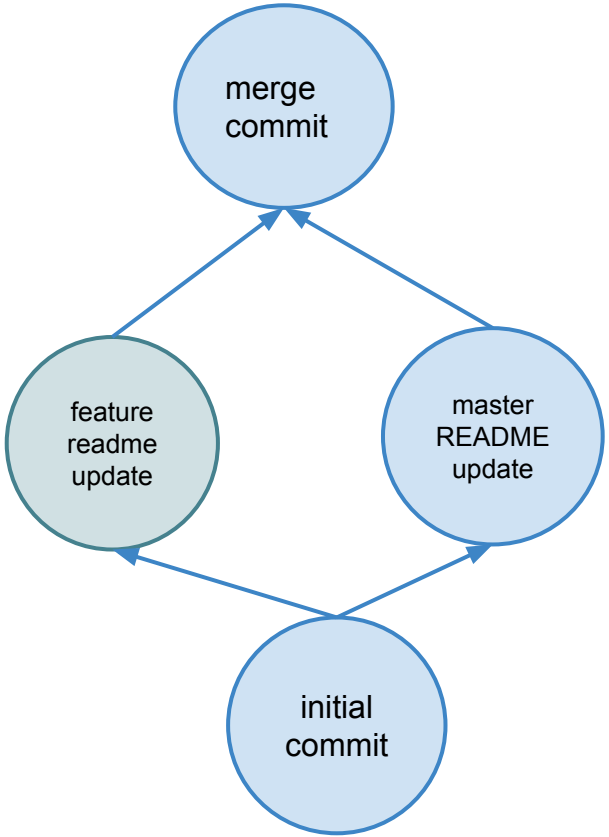
```
git merge master feature
```



14.08 Merge conflicts

```
git init
echo -e "README\n----" > README.md
git add . && git commit -m "initial commit"
git checkout -b feature
echo "This is the one and only true README
from feature branch" >> README.md
git commit -a -m "feature readme update"
git checkout master
echo "Only the master branch has the wisdom
to write a README" >> README.md
git commit -a -m "master README update"
```

```
tux@cs6server ~$: git merge feature
Auto-merging README.md
CONFLICT (content): Merge conflict in
README.md
Automatic merge failed; fix conflicts and
then commit the result.
```



14.08 Merge conflicts

⇒ git complains for `git merge master feature` about a merge conflict and that we should fix it.

Tip: you can abort a merge via `git merge --abort`

⇒ for text files, git adds annotations to mark conflicts

README.md

```
README
----
<<<<<<< HEAD
Only the master branch has the wisdom to write a README
=======
This is the one and only true README from feature branch
>>>>>>> feature
```

HEAD of the branch we are merging into

feature is the branch that is merged in

14.09 Basic conflict resolution

We have 3 options to resolve a merge conflict:

1. take the version of feature
   ```
   git checkout --theirs file
   ```
2. take the version of master
   ```
   git checkout --ours file
   ```
3. manually create a merged file

After all conflicts are resolved, create a merge commit via git commit.

⇒ more advanced conflict resolution next lecture.

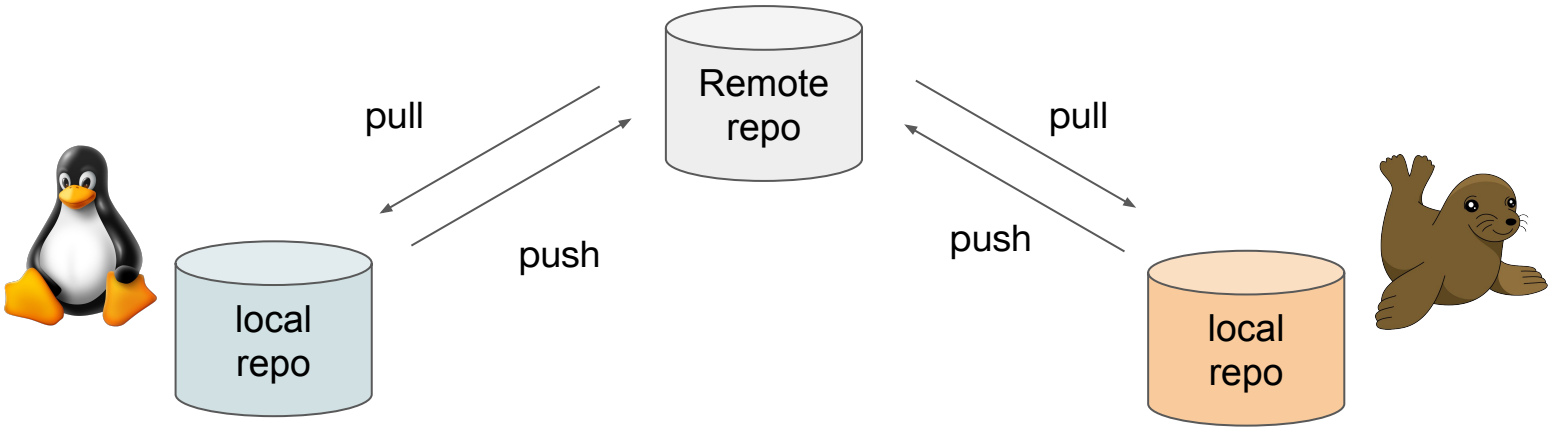
Collaboration

14.10 Remotes

⇒ to collaborate with others, need to share changes

⇒ for this remotes are used, i.e. servers which host a remote version of the repo

⇒ **pull** changes from remote or **push** changes to remote



14.10 Remotes

⇒ remote repository are typically hosted in the cloud

⇒ There are several popular repository hosting platforms like

Github	(Microsoft)
Bitbucket	(Atlassian)
GitLab	(GitLab Inc.)

⇒ Most open-source projects are on one of these platforms

14.10 Configuring remotes

`git remote add <remote_name> <remote_repo_url>`

→ maps a remote repository at remote_repo_url to remote_name

⇒ typically origin as name is used if you have a single remote.

Example:

```
git remote add origin https://github.com/cs6tux/lec12.git
```

⇒ `git remote -v` lists available remotes

14.10 Retrieving a remote repository for the first time

⇒ Typical workflow: Go to your favourite hosting platform and create a new remote repo and initialize it with a single file.

⇒ you can also clone the remote repository to get a local copy via `git clone`.

Note: You can clone via password based authentication OR use SSH keys!
⇒ SSH keys allow you to work faster!

⇒ when git clone is used, this sets up a remote called origin automatically pointing to the remote from where the repository was cloned from

14.10 Pushing branches

```
git push -u remote_name branch_name
```

⇒ pushes local branch to remote branch called branch_name
→ if empty, creates remote branch based on current one

⇒ `--set-upstream` is the long option for `-u`

⇒ this automatically sets up tracking for that branch

If you want to use a different user, you can disable the credential helper via `git config --local credential.helper ""`

14.10 Pulling branches

⇒ To list available remote branches, use `git branch -r`

⇒ remote branches can be fetched via `git fetch`

⇒ to pull changes on one branch (which tracks a remote branch) use `git pull`

⇒ you can checkout a remote branch by using `git checkout origin/master` - this creates a detached HEAD.

⇒ more convenient:

1. `git checkout -b mybranch origin/feature`
2. `git checkout --track origin/feature`

create `mybranch` which has upstream `origin/feature`

create local branch `feature` which has upstream `origin/feature`

14.11 Summary

Typical steps when working on a branch:

```
git checkout branch                     # switch to branch you want to work on
...edit files…
git status                              # check what files are modified
git add .                               # add changes
git status                              # check which files are staged
git commit -m "write a message"         # commit changes
git pull                                # fetch any changes from remote branch
… merge or rebase …
git push                                # push changes to remote
```

14.11 Final words

End of lecture.

Exam next Tue, 4pm-5:20pm @ CIT 477