Have you heard of version control?

I'm sure you know problems like this:

- Your code has an error. The fix seems obvious but unfortunately it leads to new problems. Step by step you go along a chain of small code changes until nothing works anymore. Now you're lost.
- You realize that the structure of your program is a dead-end. There are new features that you would like to add but the implementation will be very messy if you integrate it into the current code. You come up with a better architecture and start implementing it. First step: Create a copy of the old code as a reference. Then another copy of some intermediate solution. Then another copy ....
- It's Monday. C++ does not compile on Mondays. This is a mystery, since yesterday evening everything was still fine. Maybe you changed one last thing before going to bed, but there's no way to recall what that was.

There's a solution for these problems, it's called Version Control.
Version control tools are meant for managing versions of textfiles. Key features are:

- create a snapshot of your code
- compare snapshots against each other, show all the lines that are different
- undo changes
- distribute your snapshots to other team members and combine the code

There are two common systems: svn and git. Today, git is much more widely used.

Version control is an essential tool for ALL programmers. No matter if you end up programming in C++, Python or Javascript: In any professional environment, you will use version control.

Git crash course

The code in this crash course should be executed line by line, in the terminal.
You can copy and paste code with `ctrl + shift + c` and `ctrl + shift + v`

The following code will create a new folder and initialize git.
A hidden .git directory is created:

```
mkdir my_repo
cd my_repo
git init # only this command is about git
ls -la
```

Before you actually use git, it is a good idea to set up some global configuration.
It will be used to identify you as the author of code changes:

```
git config --global user.name "your name"
git config --global user.email "first.last@epfl.ch"
```

Now let's add some code and work on it.
If you want your edits to be tracked by version control, you need to `add` them.
In a second step, you can record a snapshot of your code, called a `commit` .
Please note that I'm supplying a message (like a description) for the commit with the -m flag.

```
touch README.md
nano README.md

git add README.md
git commit -m 'first commit'
```

Try and repeat the code above, with new edits to README.
I've edited my README to look like this:

```
first commit

second commit

third commit
```

After each line, I saved the file and used `git add` and `git commit -m 'xxx'` , to create a new commit.
This creates a history of commits.
You can look at this history with `git log` The output can be a bit overwhelming, it gives you a long hash, which is the name of the commit and it displays the author and commit message. Also, please note that `git log` will enter a special mode in the terminal, which you can leave by pressing **q**.

```
lhk@pop-os:~/Desktop/git_tutorial$ git log
commit 3342308f5cba716a7e5ab9a13f7806e66c88feb8 (HEAD -> master)
Author: lars klein <lars.klein@rwth-aachen.de>
Date:   Wed Feb 26 13:56:41 2020 +0100

    third commit

commit c37d77194b398d9ae5d846ddba8e4ee3167434a6
Author: lars klein <lars.klein@rwth-aachen.de>
Date:   Wed Feb 26 13:56:27 2020 +0100

    second commit

:
```

The hash is used to identify a specific commit, for example when you run `git diff`.
Change your README and run the following code.
For `<commit1>` and `<commit2>` you need to insert hashes taken from git log.
Undoing changes is also easy, as long as you haven't committed them yet.

```
git diff README.md # this will show you the changes made since the last commit
git diff <commit1>  <commit2> README.md # shows the changes made between commit1 and com

git checkout README.md # resets README.md to the last commit
```

You have to be careful with the `checkout` command.
Only when you `commit` code, it is stored by git.
That makes it easy to remove changes that are not committed yet, but impossible to restore them.

At the same time, once you have committed code, it is complicated to remove it.
Ideally, what you want is one continuous history of code changes that all build on each other.

Therefore, you should be careful before committing changes. It is good practice to use `git status` before each `git commit`.
This command shows you which files have been changed, which changes will be included in the commit and which files are not tracked by git.
A change will be included in the commit, if you have called `git add`.

If you work in a team, git will help you to keep your code synchronized. As long as you don't work on the same file, this is entirely transparent.
Let's set it up for this example. Go to https://gitlab.epfl.ch/ and click on 'new project'
Follow the steps to create a blank project.

Careful: If this is your first time using the EPFL gitlab, you will also need to set a password.
This is separate from your tequila login and can be done on the gitlab website.
When you try to connect to gitlab from your command line (next code sample), they will ask you for authentication.
Your login is your epfl email address (not gaspar login) and the password you've just set.

In a blank project, gitlab already shows the commands necessary to connect your local repository.
Please execute the code found under **Push an existing folder**.
Two commands should be new to you: `git remote add ...` connects your local repository to the gitlab server.
You will only need this once. `git push -u origin master` uploads your local changes to the server.
The counterpart to `git push` is git `git pull` .

Now you know almost all commands that you will use in your git workflow:

```
git pull # download changes from your team members
#edit your code
git add A.cc
git add B.cc
git status
git commit -m 'very descriptive message'
git push
```

If all members of your team take care to work on separate files, this will work nicely.
It is possible that `git push` fails, because the server has newer commits that are not yet present in your local directory.
But this is easily fixed by executing `git pull` again.

Behind the scenes, git will keep track of all changes and combine them to form the latest state of your code.
This can fail, if two commits change the same file. You will have to resolve these **merge conflicts** yourself.
Git will tell you which files are in conflict and add special syntax to them.
Your version is included between
`<<<<<<<` and `=======` , the other version between `=======` and `>>>>>>>`

```
Let's assume that this is the content of a file tracked by git.
There is a lot of existing work here.
<<<<<<< HEAD
and then you added something new
=======
while the file was also edited by someone else
>>>>>>> xxx
here is the rest of the file
```

You now need to edit this file to remove the merge conflict (just remove the special markup and replace it with whichever code seems appropriate).
Then save it, and use `git add` and `git commit` to create a new commit.

The last topic to be covered in this tutorial are branches. They help you to keep track of different versions of your code that exist in parallel.
By default, you are on a branch called **master**, you can use `checkout` to create new branches.
Here we also introduce the `merge` command. With this command you can use git to combine two codebases.
When you execute `pull`, git automatically tries to perform a `merge` as well, to combine your local codes with the online updates.
An easy way to experiment with `merge` are local branches.
Here you can see how merge conflicts are created and resolved:

```
git checkout -b my_branch # the -b flag creates a new branch
git status # it will tell you the name of the current branch
 #edit README.md
git add README.md
git commit -m 'edited README on new branch'
git checkout master # using checkout with an existing branch
 #edit README.md
git add README.md
git commit -m 'edited README on master'
git merge my_branch # merge conflict !!!
 # edit README.md and remove the special syntax
git add README.md
git commit -m 'resolved merge conflict'
```

A few final things and then you're good to go:

Be careful not to include any binary files in your repository.
They will change when you compile your code and lead to merge conflicts.
To avoid accidentally adding binary files, you should have a special text file called `.gitignore`.
You create this file at the root of your code directory and `add` and `commit` it.
Inside, you can specify all files that should be ignored by git.

For example, you could use this .gitignore file for C++:
https://github.com/github/gitignore/blob/master/C%2B%2B.gitignore

You will probably quickly find that there is a `-f` or `--force` flag for many commands.
For example `git push -f` prevents all those annoying merge conflicts (by overwriting the changes of your team members ...)

Do make a habit of using `git status` . It can save you a lot of time by preventing wrong commits.

To add all changes to files that are already tracked by git, use `git add -u .` .
Avoid using commands like `git add -A` or `git add *`