Laboratory 01

Alberto Artoni

23/02/2024

Introduction to `Bash` and `git`

The requirements for the labs are described in 00-environment_setup/README.md.

At the end of the installation process you should have access to a Linux-base distribution with the `mk-modules`.

Bash

What is a shell?

*"A shell is a program that provides the traditional, **text-only** user interface for Linux and other Unix-like operating systems. Its primary function is to read commands that are typed into a console [...] and then execute (i.e., run) them. The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an **interface** between the **user** and the internal parts of the **OS** (at the very core of which is the kernel)."*

From http://www.linfo.org/shell.html
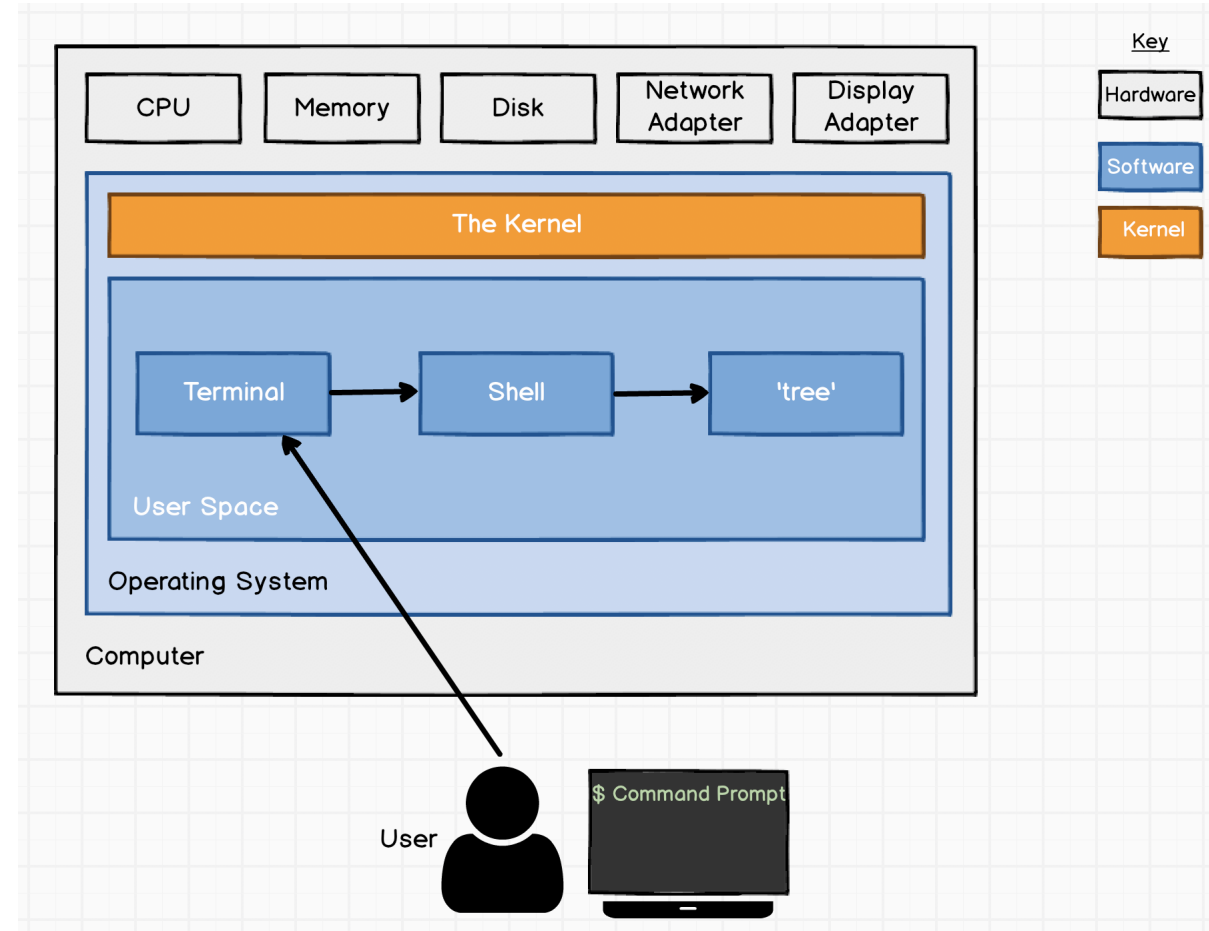
What is `Bash`?

- It stands for `Bourne Again Shell`, a homage to its creator Stephen Bourne.

- It is the *default* shell for most Unix systems and Linux distributions.

- It is both a *command interpreter* and a *scripting language*.

Remark: macOS has replaced it with zsh, which is mostly compatible with `Bash`, since v10.15 Catalina.

Variables and Environmental Variables

- Variable assignment: `A=1` **(no spaces!)**.

- Print a variable: `echo ${A}`.

Variables that affect the behaviour of the operating system are called **environmental variables**.
Many environmental variables are set by default:

- `echo ${HOME}` returns the user home directory.

Types of shell - Login

The first distinction that we make is between:

- A **login** shell logs you into the system as a specific user (it requires username and password).
- A **non-login** shell is executed without logging in (it requires a current logged in user). When you open a graphic terminal it is a non-login (interactive) shell.

Types of shell - Interactive

The second distinction we make is between:

- An **interactive** shell (login or non-login) you can interactively type or interrupt commands. For example a graphic terminal (non-login) or a virtual terminal (login). In an interactive shell the prompt variable must be set (`$PS1`).
- A **non-interactive** (sub)shell is usually run from an automated process. Input and output are not exposed (unless explicitly handled by the calling process). This is normally a non-login shell, because the calling user has logged in already. A shell running a script is always a non-interactive shell (but the script can emulate an interactive shell by prompting the user to input values).

Examples of types of shell

- `login interactive shell` : usually after you log into a remote system.
- `interactive non-login shell` : running a script explictely from the shell.
- `non-interactive non login shell` : running a command remotely withouth loggin in.
- `non-interactive login shell` : runs programs after login withouth the use interaction.

More details in Difference between Login Shell and Non-Login Shell?.

`Bash` as a command line interpreter

When running a terminal in a Unix system, it first launches the shell interpreter specified in the `SHELL` **environment variable**. If `SHELL` is unset it uses the system default.

After having *sourced* the **initialization files**, the interpreter shows the **prompt** (defined by the environment variable `$PS1`).

Initialization files are hidden files stored in the user's home directory, executed as soon as an **interactive** shell is run.

Initialization files for shells (1/2)

- **login**:
  - `/etc/profile`, `/etc/profile.d/*`, `~/.profile` for Bourne-compatible shells
  - `~/.bash_profile` (or `~/.bash_login`) for `Bash`
  - `/etc/zprofile`, `~/.zprofile` for `zsh`
  - `/etc/csh.login`, `~/.login` for `csh`
- **non-login**:
  - `/etc/bash.bashrc`, `~/.bashrc` for `Bash`

Initialization files for login shells (2/2)

- **interactive**:
  - `/etc/profile`, `/etc/profile.d/*` and `~/.profile`
  - `/etc/bash.bashrc`, `~/.bashrc` for `Bash`

- **non-interactive**:
  - `/etc/bash.bashrc` for `Bash` (but most of the times the script begins with: `[ -z "$PS1" ] && return`, *i.e.* don't do anything if it's a non-interactive shell).
  - depending on the shell, the file specified in `$ENV` (or `$BASH_ENV`) might be read.

Getting started

To get a little hang of the bash, let's try a few simple commands:

- `echo` : displays a string or a variable.

- `date` : displays the current time and date.

- `clear` : clean the terminal. (also `Ctrl+l`)

Basic Bash Commands 1/2

- `pwd` stands for **Print working directory**. The command displays the current path of the shell.
- `ls` stands for a **List** and it lists the content of a directory.
- `cd` stands for **Change directory** and changes the active directory to the path specified.

Example:

- `cd ..` change path to the above directory.
- `ls .` list files in the current directory.
- `cd -` change directory to the previous path.

Basic Bash Commands 2/2

- `mkdir` stands for **Make directory** and it is used to make a new directory or a folder.
- `mv` stands for **Move** and it moves one or more files or directories from one place to another. It is also used to *rename* files.
- `touch` command is used to create new, empty files. It is also used to change the timestamps on existing files and directories.
- `rm` stands for **Remove** and it removes files.
- `export` is employed to export variables to child processes.

Example

- `rm -r myDir` removes the directory `myDir`.
- `mv temp.txt tmp.txt` renames a file.

Not all commands are equal

When executing a command a subprocess is created. A subprocess inherits all the environment variables from the parent process, executes the command and returns the control to the calling process.

A subprocess cannot change the state of the calling process.

The command `source script_file` executes the commands contained in `script_file` as if they were typed directly on the terminal. Usually it is employed for scripts that have to change environmental variables or define aliases or bash function.

Run a script

To run a script you need to change the access permissions of the file. To make a file executable run:

```
chmod +x script_file
```

Finally, remember that the **first line of the script** tells the shell which interpreter to use while executing the file. For example, if your script starts with `#!/bin/bash` it will be run by `Bash`, if is starts with `#!/usr/bin/env python` it will be run by `python`.

Built-in commands

Some commands, like `cd` are executed directly by the shell, without creating a subprocess.

Indeed it would be impossible the have `cd` as a regular command! Why?

Answer: a subprocess cannot change the state of the calling process, whereas `cd` needs to change the value of the environmental variable `PWD` (that contains the name of the current working directory).

Other commands

In general a *command* can refer to:

- a builtin command;
- an executable;
- a function.

The shell looks for executables with a given name within directories specified in the environment variable `PATH`, whereas aliases and functions are usually sourced by the `.bashrc` file (or equivalent).

Aliases

The usual commands we employ are aliases with specific options.
For instance:

```
alias ls='ls --color=auto'
```

To display the full alias of `command_name` write:

```
type command_name
```

To check its location type:

```
which command_name
```

A warning about filenames

In order to live happily and without worries, **don't** use spaces nor accented characters in filenames! The space character is employed as a separation charachter.

Choose a standard practice

- *snake case*: `my_wonderful_file_name`.

- *camel case*: `myWonderfulFileName`.

- *screaming snake case*: `MY_WONDERFUL_FILE_NAME`.

NEVER write `my wonderful file name`.

Wildcards

A wildcard is a symbol (or set of symbols) representing other characters.

- The wildcard `?` matches a single character.
- The wildcard `*` matches any number of characters.
- The wildcard `[]` matches characters that are enclosed in square braces.

Examples:

- `s??n` matches anything that begins with `s` and ends with `n` and has two characters between them.
- `s*n` matches anything that starts with `s` and finishes with `n`.
- `s[on]n` matches `s[on]n` will match only `son` and `snn`.
- `s[a-d]n` matches the words `san`, `sbn`, `scn`, `sdn`.

Exercises

- Go to your home folder. (*Suggestion:* you can either use `~` or `$HOME`)
- Create a folder named `test1` .
- Go inside `test1` and create a directory `test2` .
- Go inside `test2` and then up one directory.
- Create the following files `f1.txt` , `f2.txt` , `f3.dat` , `f4.md` , `readme.md` , `.hidden` .
- List all files in the directory, also the hidden ones.
- List only files with txt extension (*Suggestion:* use `*` wildcard).
- List files with `1` , `2` , `3` or `4` in the name (*Suggestion:* use `[1-4]` wildcard).
- Move the `readme.md` in `test2` .
- Move all txt files in `test2` in one command.
- Remove `f3.dat` .
- Remove all contents of `test2` and the folder itself in one commands.

Download and deflate a Matrix

With `wget` you can retrieve content from web servers. For instance, you can download a matrix from the matrix market with `wget` `https://math.nist.gov/pub/MatrixMarket2/NEP/mhd/mhd416a.mtx.gz` . To unzip the file, simply type `gzip -dk mhd416a.mtx.gz`

More commands

- `cat` stands for **Concatenate** and it reads a file and outputs its content. It can read any number of files, and hence the name concatenate.
- `wc` is short for **Word count**. It reads a list of files and generates one or more of the following statistics: newline count, word count, and byte count.
- `grep` stands for **Global regular expression print**. It searches for lines with a given string or looks for a pattern in a given input stream.
- `head` shows the first lines of a file.
- `tail` shows the last lines of a file.
- `file` reads the files specified and performs a series of tests in attempt to classify them by type.

Redirection, Pipelines and Filters

We can add operators between commands in order to chain them together.

- The pipe operator `|` , forwards the output of one command to another. E.g. `cat /etc/passwd | grep user` checks system information about `user` .
- The redirect operator `>` sends the standard output of one command to a file.
- The append operator `>>` appends the output of one command to a file.
- The operator `&>` sends the standard output and the standard error to file.
- `&&` pipe is activated only if the return status of the first command is 0.
- `||` pipe is activated only if the return status of the first command is different from 0.

Exercises

- Create a file with the current date (one command) and display its content.

- Count the number of lines in the matrix `mhd416a.mtx` (*Suggestion:* use `cat`, `wc` and `|`).

- List the entries of the matrix that are smaller than `1e-10` in absolute value. You can assume that all values are in exponential format and all values are greater than `1e-100` in absolute value. Count how many entries satisfy this criteria (*Suggestion:* use `cat`, `grep`, `wc` and `|`).

Regular Expressions

Similarly to wildcards, **regular expressions** (*regex* or *regexp*) are strings that match families of strings. They are extremely useful in extracting information from any text by searching for one or more matches of a specific search pattern. They are supported in almost all programming languages and by `grep`.

In this site you can test and debug your regex interactively:
https://regex101.com/

In this site you can find step-by-step interactive exercises:
https://regexone.com/

Regex - Anchors

- `^` : matches the start of the line

- `$` : matches the end of the line

Examples:

```
echo "the the end end" | grep --color -P ^the
echo "the the end end" | grep --color -P the
echo "the the end end" | grep --color -P end
echo "the the end end" | grep --color -P end$
```

Notes:

`--color` option colors the grep match

`-P` option is for regex

`-i` option is for case insensitive

Regex - Quantifiers

- `(...)` : characters in brackets define a **group**, meaning they are treated as a single entity by quantifiers.
- `*` : matches **zero or more** times the group before this character.
- `+` : matches **one or more** times the group before this character.
- `?` : matches **zero or one** times the group before this character.
- `{2}` : matches **exactly** two times the group before this character.
- `{2,}` : matches two or more times the group before this character.
- `{2,5}` : matches two to five times the group before this character.

Examples:

```
echo abcbcbc | grep --color -P "a(bc)?"
echo abcbcbc | grep --color -P "a(bc){2}"
echo abcbcbc | grep --color -P "a(bc)*"
```

Regex - OR operator

- `|` : matches the group on the right or on the left
- `[...]` : matches any of the characters in the square brackets
- `[^...]` : `^` inside square brackets has a different meaning, it is a negation: means any character but the ones in the square brackets

Examples:

```
echo abcbcbc | grep --color -P "[ab]"
echo abcbcbc | grep --color -P "(a|b)"
echo abcbcbc | grep --color -P "abc[^d]c"
```

Regex - Character classes

- `\d` : matches a single character that is a digit (equivalent to `[0-9]`).
- `\w` : matches a word character (alphanumeric and underscore).
- `\s` : matches a whitespace character (includes tabs and line breaks).
- `.` : matches any character.

Examples:

```
echo abcbcbc-1234 | grep --color -P "\w"
```

Notes:

To match special characters like `.` , use the escape `\` :

```
echo "ab.d-abcd" | grep --color -P "ab[^\.]d"
```

Regex - Greedy vs lazy

The quantifiers `* + {}` are **greedy** operators, so they expand the match as far as they can through the provided text.

For example, `<.+>` matches `<div>simple div</div>` in `This is a <div> simple div</div> test`. In order to catch only the div tag we can use a `?` to make it lazy:

```
echo "This is a <div> simple div</div> test" | grep --color -P "<.+>"
echo "This is a <div> simple div</div> test" | grep --color -P "<.+?>"
```

Regex - Look ahead

- `d(?=r)` matches a `d` only if is followed by `r`, but `r` will not be part of the overall regex match.

- `(?<=r)d` matches a `d` only if is preceded by an `r`, but `r` will not be part of the overall regex match.

- `d(?!r)` matches a `d` only if is not followed by `r`, but `r` will not be part of the overall regex match.

- `(?<!r)d` matches a `d` only if is not preceded by an `r`, but `r` will not be part of the overall regex match.

Advanced commands - `tr`

- `tr` stands for **translate**. It supports a range of transformations including uppercase to lowercase, squeezing repeating characters, deleting specific characters, and basic find and replace. For instance:
  - `echo "Welcome to apsc labs" | tr [a-z] [A-Z]` converts all characters to upper case.
  - `echo -e "A;B;c\n1,2;1,4;1,8" | tr "," "." | tr ";" ","` translates a line of a CSV in italian format to a standard format.
  - `echo "my ID is 73535" | tr -d [:digit:]` deletes all the digits from the string.

Advanced commands - `sed`

- `sed` stands for **stream editor** and it can perform lots of functions on file like searching, find and replace, insertion or deletion. We give just an hint of its true power:
    - `echo "unix is great os. unix is open source." | sed 's/unix/linux/'` replaces the first occurrence of "unix" with "linux".
    - `echo "unix is great os. unix is open source." | sed 's/unix/linux/2'` replaces the second occurrence of "unix" with "linux".
    - `echo "unix is great os. unix is open source." | sed 's/unix/linux/g'` replaces all occurrences of "unix" with "linux".
    - `echo -e "ABC\nabc" | sed '/abc/d'` delete a line matching "abc".
    - `echo -e "1\n2\n3\n4\n5\n6\n7\n8" | sed '3,6d'` delete lines 3 to 6.

Advanced commands - `cut` and `find`

- `cut` is a command for cutting out the sections from each line of files and writing the result to standard output.
  - `cut -b 1-3,7- state.txt` cut bytes (`-b`) from 1 to 3 and from 7 to end of the line.
  - `echo -e "A,B,C\n1.22,1.2,3\n5,6,7\n9.99999,0,0" | cut -d "," -f 1` get the first column of a CSV (`-d` specifies the delimiter among field, `-f n` specifies to pick the n-th field from each line).
- `find` is used to find files in specified directories that meet certain conditions. For example: `find . -type d -name "*lib*"` find all directories (not files) starting from the current one (`.`) whose name contain lib.
- `locate` is less powerful than `find` but much faster since it relies on a database that is updated on a daily base or manually using the command `updatedb` . For example: `locate -i foo` finds all files or directories whose name contains `foo` ignoring case.

Quotes

Double quotes may be used to identify a string where the variables are interpreted. Single quotes identify a string where variables are not interpreted. Check the output of the following commands

```
a=yes
echo "$a"
echo '$a'
```

The output of a command can be converted into a string and assigned to a variable for later reuse:

```
list=$(ls -l)
```

Processes

- Launch a command and send it in the background: `./my_command &`

- `Ctrl-Z` suspends the current subprocess and `bg` reactivates the suspended subprocess in the background.

- `jobs` lists all subprocesses running in the background in the terminal.

- `fg %n` brings back to the foreground the n-th subprocess in the background.

- `Ctrl-C` terminates the subprocess in the foreground (when not trapped).

- `kill pid` sends termination signal to the subprocess with id `pid`. You can get a list of the most computationally expensive processes with `top` and a complete list with `ps aux` (usually `ps aux` is filtered through a pipe with `grep`)

All subprocesses in the background of the terminal are terminated when the terminal is closed (unless launched with `nohup`, but that is another story...)

How to get help

Most commands provide a `-h` or `--help` flag to print a short help information:

```
find -h
```

`man command` prints the documentation manual for command.
There is also an info facility that sometimes provides more information: `info command`.
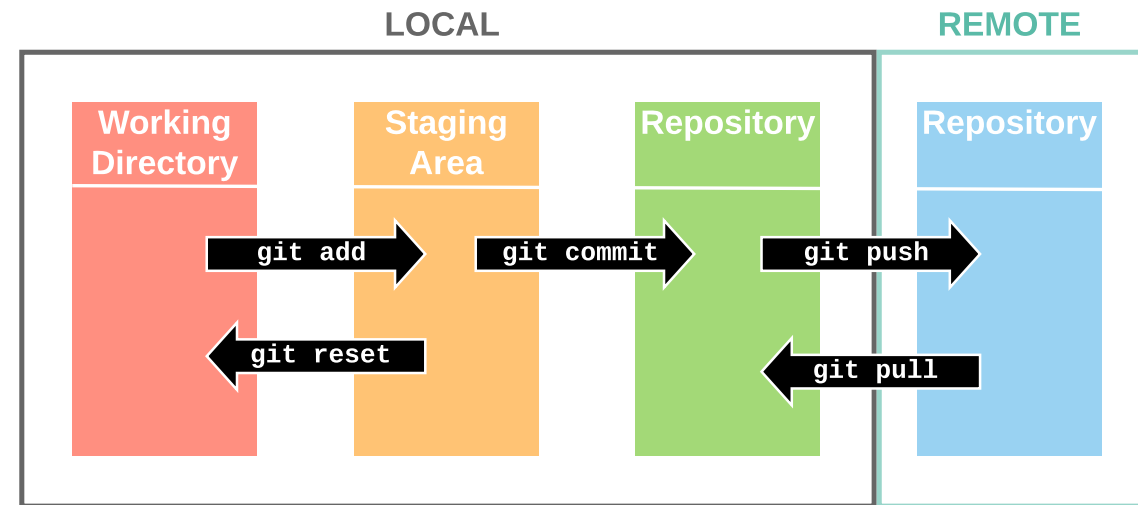
`git`

Version control

Version control, also known as source control, is the practice of tracking and managing changes to software code. Version control systems are software tools that help software teams manage changes to source code over time.

`git` is a free and open-source version control system, originally created by Linus Torvalds in 2005. Unlike older centralized version control systems such as SVN and CVS, Git is distributed: every developer has the full history of their code repository locally. This makes the initial clone of the repository slower, but subsequent operations dramatically faster.

How does `git` works?

1. Create (or find) a repository with a git hosting tool (an online platform that host you project, like Github or Gitlab)

2. `clone` (download) the repository

3. `add` a file to your local repo

4. `commit` (save) the changes, this is a local action, the remote repository (the one in the cloud) is still unchanged

5. `push` your changes, this action synchronizes your version with the one in the hosting platform

LOCAL   REMOTE

Working Directory	Staging Area	Repository	Repository
	git add →	git commit →	git push →
	← git reset		← git pull

How does `git` works? (Working in teams)

If you and your teammates work on different files the workflow is the same as before, you just have to remember to `pull` the changes that your colleagues made.

If you have to work on the same files, the best practice is to create a new `branch`, which is a particular version of the code that branches form the main one. After you have finished working on your feature you `merge` the branch into the main.

Other useful `git` commands

- `diff` show the differences between your code and the last commit
- `status` list the status of all the files (e.g. which files have been changed, which are new, which are deleted and which have been added)
- `log` show the history of commits
- `checkout` switch to a specific commit or brach
- `stash` temporarily hide all the modified tracked files

SSH authentication

1. Register to Github.

2. Create the SSH key.

3. Add it to your account.

4. Configure your machine:

```
git config --global \
  user.name "Name Surname"
git config --global \
  user.email "name.surname@email.com"
```

See here for more details on SSH.
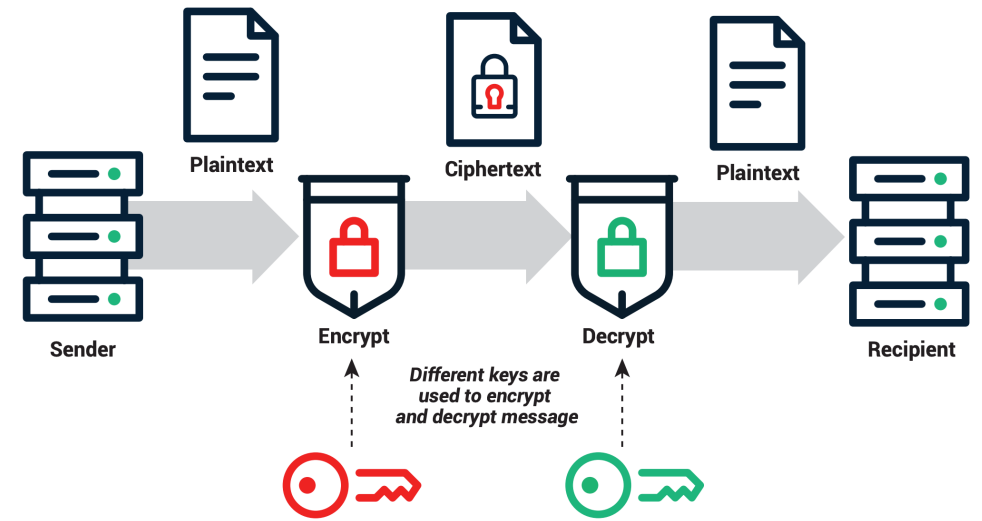
Hands on `git` - Working on different files (1/2)

1. Get in groups of 2-3 students

2. One of the members of the group creates a new repo (go to https://github.com/ and click the plus in the top right corner) and everyone clone it

3. Everyone should create a file with its name and put it online, after everyone is done pull the latest version

Hands on `git` - Working on the same file (2/2)

Now try working on the same file: every one should create a hello world `main.cpp` that shows a greet with your name. To avoid conflicts:

1. create a different branch with `git checkout -b [name_of_your_new_branch]`

2. develop your code and put your branch online

3. when everyone is done merge your branch into main with the following commands

```
git checkout main
git pull origin main
git merge [name_of_your_new_branch]
git push origin main
```

the first one to do so will have no issue, the following will have to solve a merge conflict and commit and push your "solution" to the conflict

Hands on `git` - The course repository

1. Clone the APSC repository:

```
git clone --recursive  git@github.com:pacs-course/pacs-examples.git
```

2. Go to `pacs/Examples` and follow the instructions.

3. Go to `pacs/Examples/src/Utilities` and run

```
make
make install
```

4. **Have fun!**