Programming and Laboratory - 2

1. Introduction

Giulio Ermanno Pibiri — giulioermanno.pibiri@unive.it
Nicola Prezza — nicola.prezza@unive.it
Department of Environmental Sciences, Informatics and Statistics

Overview

- Course introduction and general information:
  - course structure
  - program
  - material
  - exam
- Setting-up a development environment:
  - terminal
  - compilation

What is this course?

Welcome to CT0442 — “Programming and Laboratory — Mod. 2”.

This is an introductory course about C++: **fundamentals + coding** (debugging and exercises).

C++ fundamentals:

- generic programming (templates)
- copy/move semantics
- function/operator overloading
- C++ standard template library (STL): containers, iterators, algorithms

Coding

- Do the exercises!
- To be tattooed on your arm: **practice makes you better**.
- Method:
    1. Study some techniques.
    2. Read someone else’s code and try to understand it.
    3. Try to write something on your own.
    4. Repeat.
- Approach: **be skeptic**. Verify everything taught in class by your own.

Tentative program breakdown (might change)

- 48h course — 6 CFU [24 lectures]
- Topics:
  1. Introduction and tools — shell, make, git, valgrind, sanitizers [5 lectures]
  2. Warm-up on recursion [2 lectures]
  3. Advanced recursion — parsers and I/O streams [4 lectures]
  4. Templates [4 lectures]
  5. Containers - copy/move semantics, operator overloading, iterators [6 lectures]
  6. Standard Template Library (STL) [3 lectures]

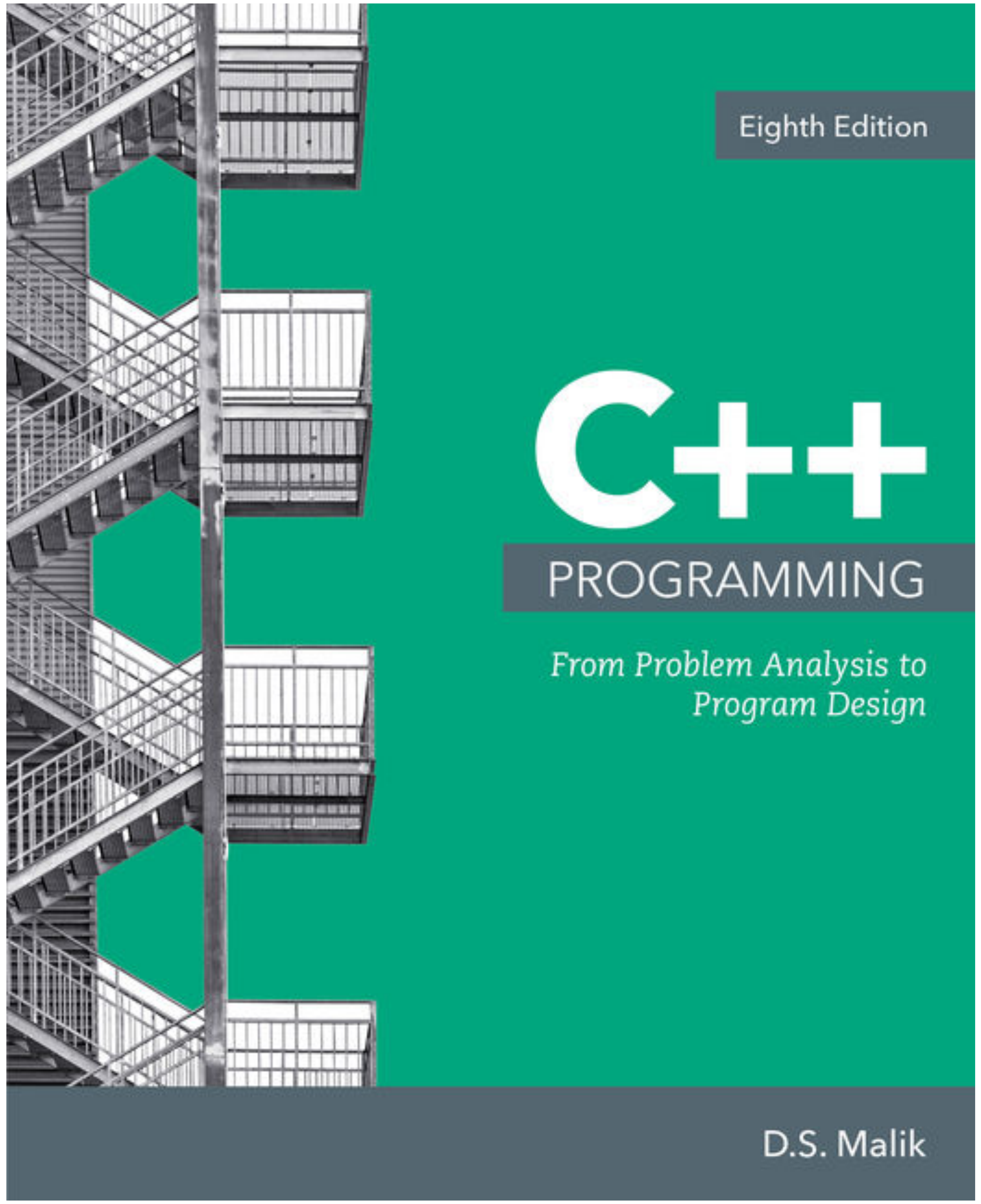
Lectures

- Lectures are only given **in presence**.
- Video-recordings only for students with special needs (I cannot decide who falls in this category, so don’t bother asking me if I can give you the recording…).
- Very important to participate actively during the lectures!
- All the material we will see is part of the exam (written, oral, and project).
- **Important:** subscribe to the **Moodle** page for course material and communications.

Material

- Slides and code on Moodle.
- Participate actively to the lectures: practice is fundamental!
- Book:

  D.S. Malik: *C++ Programming: From Problem Analysis to Program Design*. Eight Ed. Cengage Learning. 2018



Exam workflow

How to pass the exam? Written test + oral test + weekly exercises + project.

- Pass Module 1:
    1. **Written test.**
    2. **Oral test**. Can be done only if written test has been passed. If failed, you have to **repeat** also the written test.
- Pass **Module 2**:
    1. **Weekly exercises**. Mandatory!
    2. **Project**. Will be evaluated only if at least 60% of weekly exercises of the current year have been passed.

Exam rules

- **Module 1 counts 70% on the final evaluation.**
- **Module 2 (the project) counts 30%.**
- Evaluations of Module 1 and Module 2 remain valid for 1 year from when they are passed.
- The project will be described and assigned towards the end of the course (in May).
- You will have **4** trials (i.e., “appelli”) to submit the project for evaluation: 1 in June, 1 in July, 1 in September, and 1 in January.
- **After January, you will have to follow a new project specification (relative to the next academic year).**

Exception: part-time/working students

- All the same, except the weekly exercises.
- **Weekly exercises**. If you prefer, these can be covered with questions during the **oral test**. In this case, you do not need to solve the weekly exercises.
- Let us know **via email** if you prefer this solution.

Module 2 exam — individual project

- You'll have to implement a **C++ container** (more details during the course) solving a non-trivial task.
- Projects of the previous years:
  - https://github.com/PEL-unive/progetto-22-23
  - https://github.com/PEL-unive/progetto-23-24
- Remember: all implemented functions will be tested thoroughly (move/copy semantics, constructors, destructor, memory management, operator overloading, Iterators, I/O, etc).

  You will have to run thorough tests on your project!

  (Projects often fail because they have not been tested properly).

Module 2 exam — individual project

Do not commit plagiarism!

- Every year, several students copy someone else's project (plagiarism).
- We use a **plagiarism-detector** on the projects of all sessions in the academic year.
- **We take this seriously**. What if you commit plagiarism?
- If the plagiarism is clear:
    1. You have to **repeat the exam next year**. Module 1 is canceled, if you already passed it.
    2. Ca' Foscari's disciplinary committee is notified. You risk to be **expelled** from the university (the same happens if you copy in any other exam).
- If there is only a suspect of plagiarism: there will be an **oral exam on the project**.

Module 2 exam — individual project

Recommendations to avoid plagiarism:

- Discussing the high-level ideas for solving the project is ok, but **no code has to be shared** between students. If you share code, be sure that our plagiarism detector will find it out.

- **Do not post your code in a public online repository**, or in any place where other students can find it. Some other student will find it and copy it (this happened in the past).

- **Do not copy-paste code snippets** from the internet. This is also plagiarism!

Questions

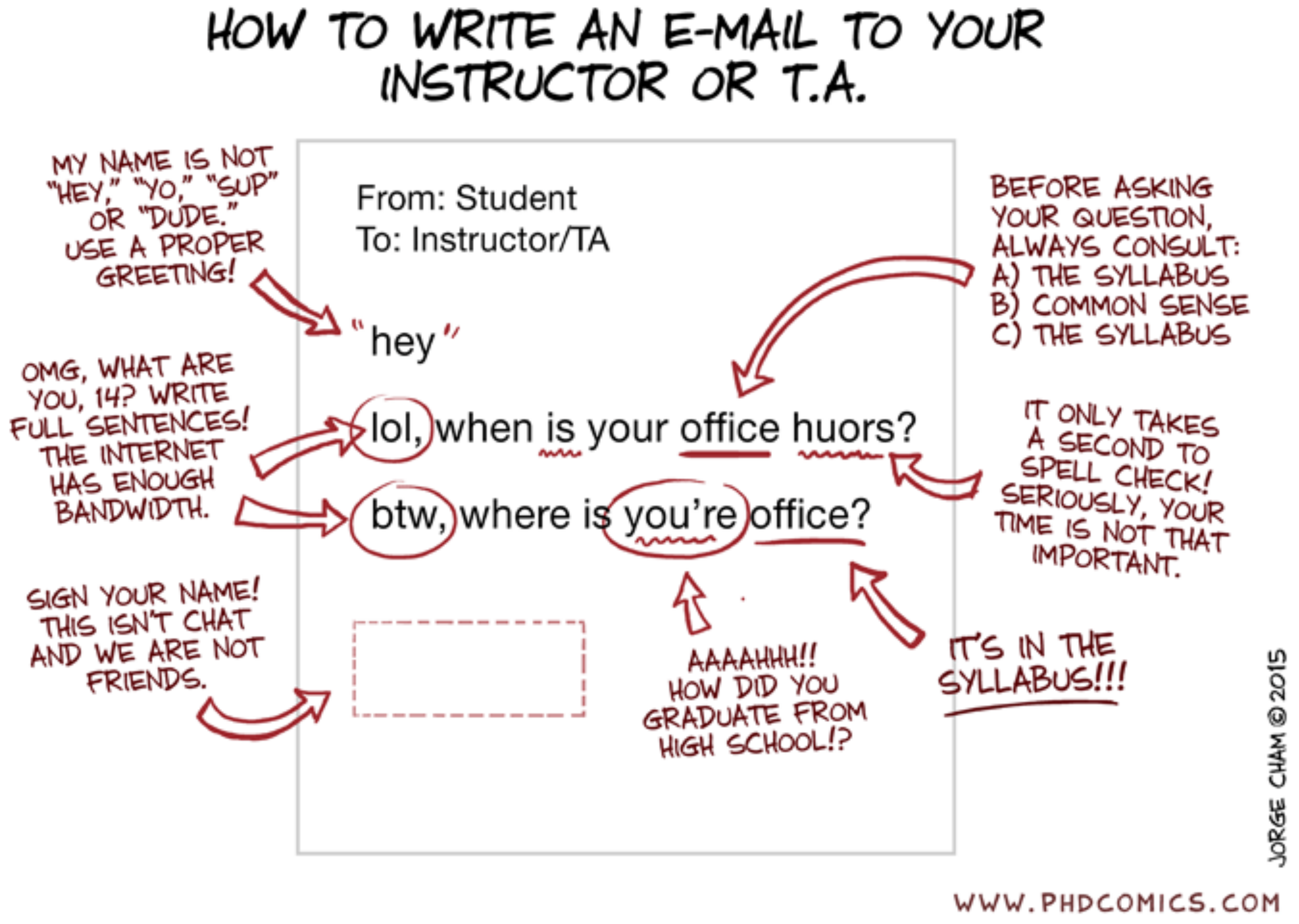
You’re kindly invited to ask questions at any time!

Follow this workflow:

1. Ask your question on the **Moodle forum**. Stimulate participation.
2. Only if nobody answers, send me an email.
3. You can also ask for a meeting (ricevimento), either on-line or in my office.

   Send me an email to fix a date for a meeting.

E-Mails


HOW TO WRITE AN E-MAIL TO YOUR INSTRUCTOR OR T.A.
MY NAME IS NOT "HEY," "YO," "SUP" OR "DUDE." USE A PROPER GREETING!
From: Student
To: Instructor/TA
"hey"
lol, when is your office huors?
btw, where is you're office?
OMG, WHAT ARE YOU, 14? WRITE FULL SENTENCES! THE INTERNET HAS ENOUGH BANDWIDTH.
SIGN YOUR NAME! THIS ISN'T CHAT AND WE ARE NOT FRIENDS.
BEFORE ASKING YOUR QUESTION, ALWAYS CONSULT: A) THE SYLLABUS B) COMMON SENSE C) THE SYLLABUS
IT ONLY TAKES A SECOND TO SPELL CHECK! SERIOUSLY, YOUR TIME IS NOT THAT IMPORTANT.
AAAAHHH!! HOW DID YOU GRADUATE FROM HIGH SCHOOL!?
IT'S IN THE SYLLABUS!!!
JORGE CHAM © 2015
WWW.PHDCOMICS.COM

https://marktomforde.com/academic/undergraduates/Email-Etiquette.html

The C++ programming language

The C++ programming language

- C++ is a **high-level**, **general-purpose**, **compiled** programming language.
- Very powerful: there is literally nothing that cannot be done in C++.
- False myth: “C++ is fast”.
  No. C++ gives you control over performance.

The C++ programming language

- C++ is a **high-level**, **general-purpose**, **compiled** programming language.
- Very powerful: there is literally nothing that cannot be done in C++.
- False myth: “C++ is fast”.
  No. C++ gives you control over performance.
- Originally developed by Bjarne Stroustrup while at Bell Labs in 1983.
- At the beginning: “C with classes”.
- A lot of further development, e.g. type abstraction (templates), STL, etc.
- Standardized in 1998 (C++98). Following standards (major revisions):

  C++03, **C++11** (we will use this), C++14, C++17, C++20

Bjarne Stroustrup

The C++ programming language

The 1998 standard consists in two parts:

1. The language core;

2. The standard library (stdlib), which includes most of the **Standard Template Library (STL)**.

STL was originally a third-party library, developed by Alexander Stepanov (HP and Silicon Graphics), before it was included in the C++ standard. (More details later in the course.)

Most compilers (including g++) include an implementation of the standard library.

Alexander Stepanov

C++ vs. C

C++ originated as a C fork, but:

- C is not a subset of C++.

  Some C code does not compile in C++ (or it compiles but then has subtle logical errors).

- Almost no C++ code compiles as C code. If it does…then you are not programming with proper C++!

More info: https://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B

Some useful on-line resources

- **Reference**
    - https://cplusplus.com/reference/
    - https://en.cppreference.com/w/
- **Cheat-sheets**
    - https://github.com/AnthonyCalandra/modern-cpp-features/blob/master/CPP11.md
    - https://github.com/cpp-best-practices/cppbestpractices/blob/master/00-Table_of_Contents.md
- **Advanced**
    - Agner Fog's optimisation guide: https://www.agner.org/optimize/optimizing_cpp.pdf
    - Hardware effects: https://github.com/Kobzol/hardware-effects
- **CppCon** YouTube channel: https://www.youtube.com/@CppCon
- **Performance blogs**
    - https://lemire.me/blog/
    - https://travisdowns.github.io/

Setting-up a development environment

Setting-up a development environment

We suggest you to use the Linux operating system.

(Windows treats I/O differently and if you compile your exam project on Windows you may have troubles – it happens to several students every year).

We will compile and try your project code on Linux anyway, so…

Setting-up a development environment

We suggest you to use the Linux operating system.

(Windows treats I/O differently and if you compile your exam project on Windows you may have troubles – it happens to several students every year).

We will compile and try your project code on Linux anyway, so…

Some **alternatives**:

- Install Ubuntu on your PC (e.g. dual boot). Different procedures for different PCs, ask on the forum if you need help. Not needed if you have a Mac (unix-based, similar terminal environment).

- Windows Subsystem for Linux (WSL): https://ubuntu.com/tutorials/install-ubuntu-on-wsl2-on-windows-11-with-gui-support#1-overview. **Suggested solution if you have Windows.**

- Create an Ubuntu virtual machine (https://www.youtube.com/watch?v=x5MhydijWmc).

Setting-up a development environment

Creating an Ubuntu virtual machine with Virtual Box:

1. First install Oracle VirtualBox — https://www.virtualbox.org/wiki/Downloads

2. If you use Windows, download “Windows hosts” from the link above (or, if you have a MAC, “OSX hosts”, either Intel or ARM depending on your processor: ask me if you do not know!) and execute the file.

3. Proceed with the default installation.

4. Run VirtualBox.

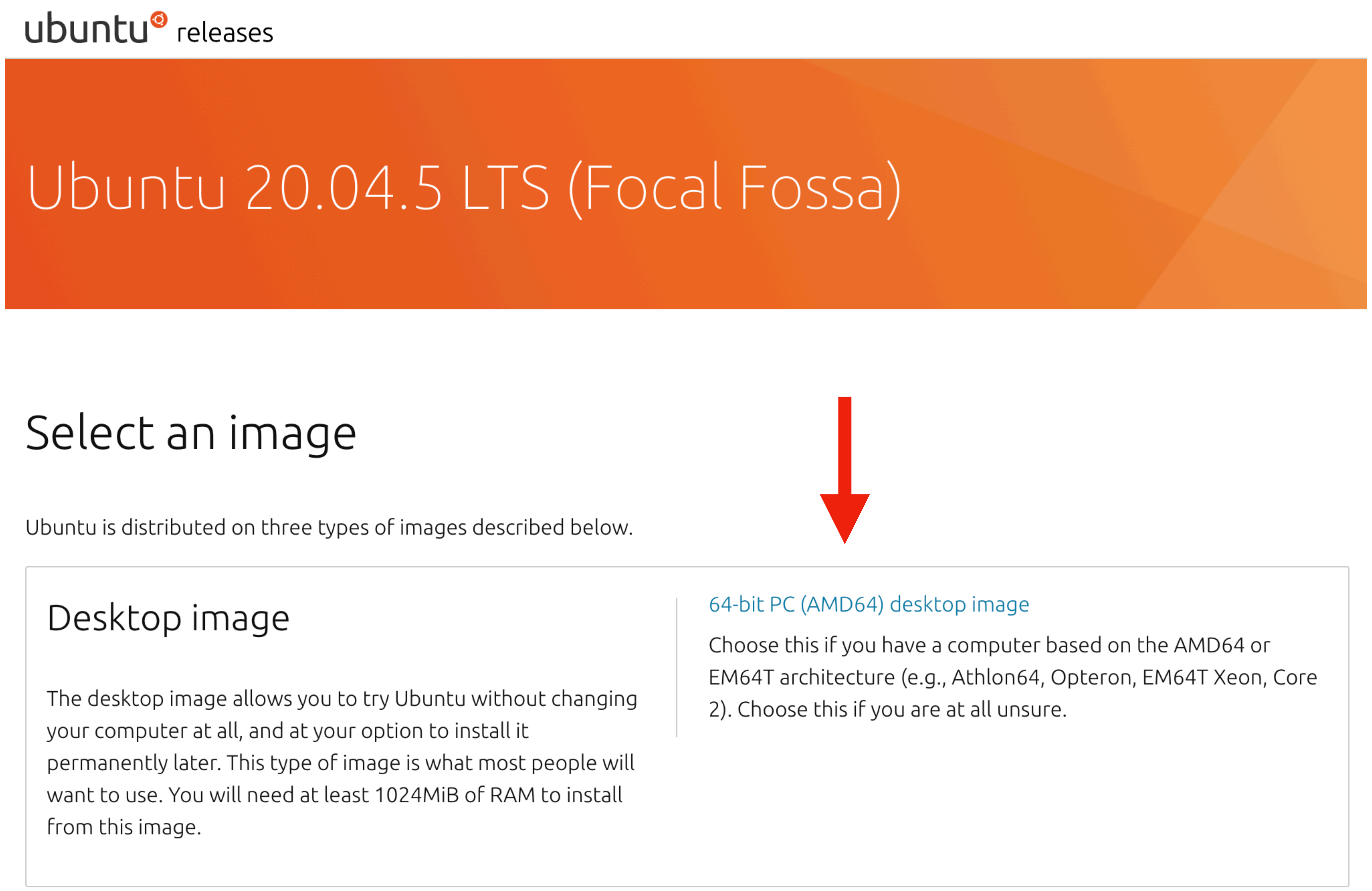
5. Install Ubuntu 20-04 LTS (Focal Fossa) — follow this tutorial: https://linuxhint.com/install_ubuntu_virtualbox_2004/

Install Ubuntu 20-04 LTS (Focal Fossa)

- Essentially two steps:

  1. Download the iso file (~3.9GB) from http://releases.ubuntu.com/20.04/.

  2. When creating a new virtual machine on Virtual Box, specify the downloaded iso file.



Ubuntu 20.04.5 LTS (Focal Fossa)

Select an image

Ubuntu is distributed on three types of images described below.

Desktop image

The desktop image allows you to try Ubuntu without changing your computer at all, and at your option to install it permanently later. This type of image is what most people will want to use. You will need at least 1024MiB of RAM to install from this image.

64-bit PC (AMD64) desktop image

Choose this if you have a computer based on the AMD64 or EM64T architecture (e.g., Athlon64, Opteron, EM64T Xeon, Core 2). Choose this if you are at all unsure.

Setting-up a development environment

- We will use a minimalist setup. **Abstract away technology and focus on C++ concepts.**
- Two things:
  1. **a text editor** (my preference, Sublime Text, https://www.sublimetext.com/3) to write the code;
  2. **the terminal** to **compile and run** the code.
- A text editor is an application that must be installed first (see next).
- The terminal is another application that is already available in Linux, as well as the compiler (you do not need to install anything).

Terminal: using a Linux shell

Linux shell

- A **shell** is a user interface to **access the operating system**'s services.
- The **Terminal** is an application providing such user interface.
- We will learn how to compile a C++ program from the shell.
- In order to do that, we need some basic notion of the shell itself (mainly filesystem navigation and basic file manipulation).

Graphical vs. textual interface

- All systems have a set of graphical applications (word processor, email reader, internet browser, etc.) that can be controlled using mouse and keyboard.
- All systems can **also** be controlled using a “textual” interface.

Interface	**Pros**	**Cons**
Graphical	Easy to learn	Cannot be automatized
Textual	Easy to automatize	Hard to learn

Graphical vs. textual interface

- The **prompt** includes information like the username, the name of the computer, and the current directory.
- Example.

```
user-name@computer-name:~$
```

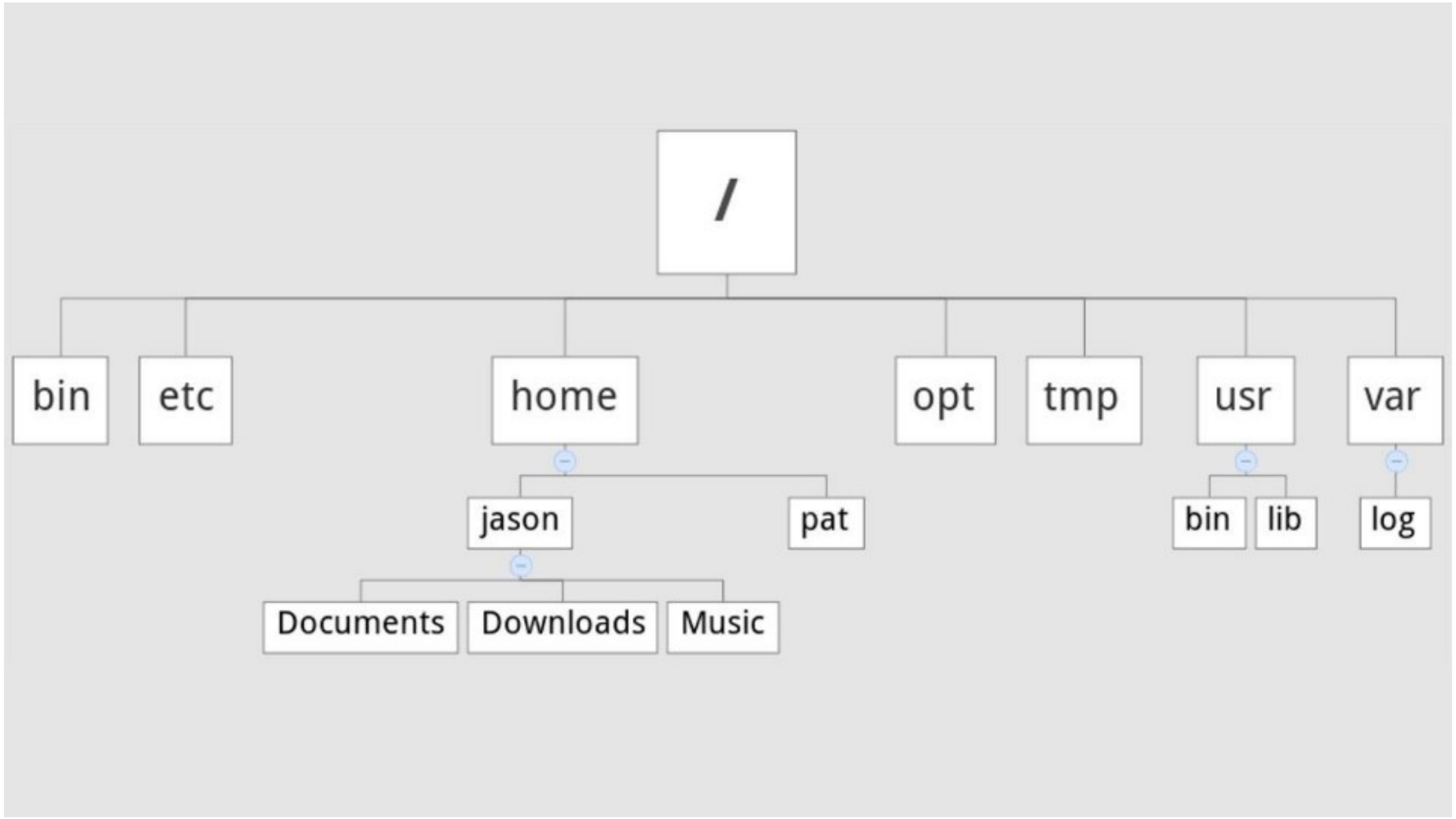
means:

- user name → “`user-name`”
- computer name → “`computer-name`”
- filesystem position → ~ (“home” directory)
- Characters `@` : `$` are separators. To the right of `$` we write our commands.

Filesystem

- A “filesystem” is a hierarchic representation (a tree) of a set of files.
- Files are organized into folders (directories).
- Folders can be nested into sub-folders.
- Each file and folder has a name and a path (the path from the root to the object).
- The “root” directory has no name and it is represented as `/` (slash).

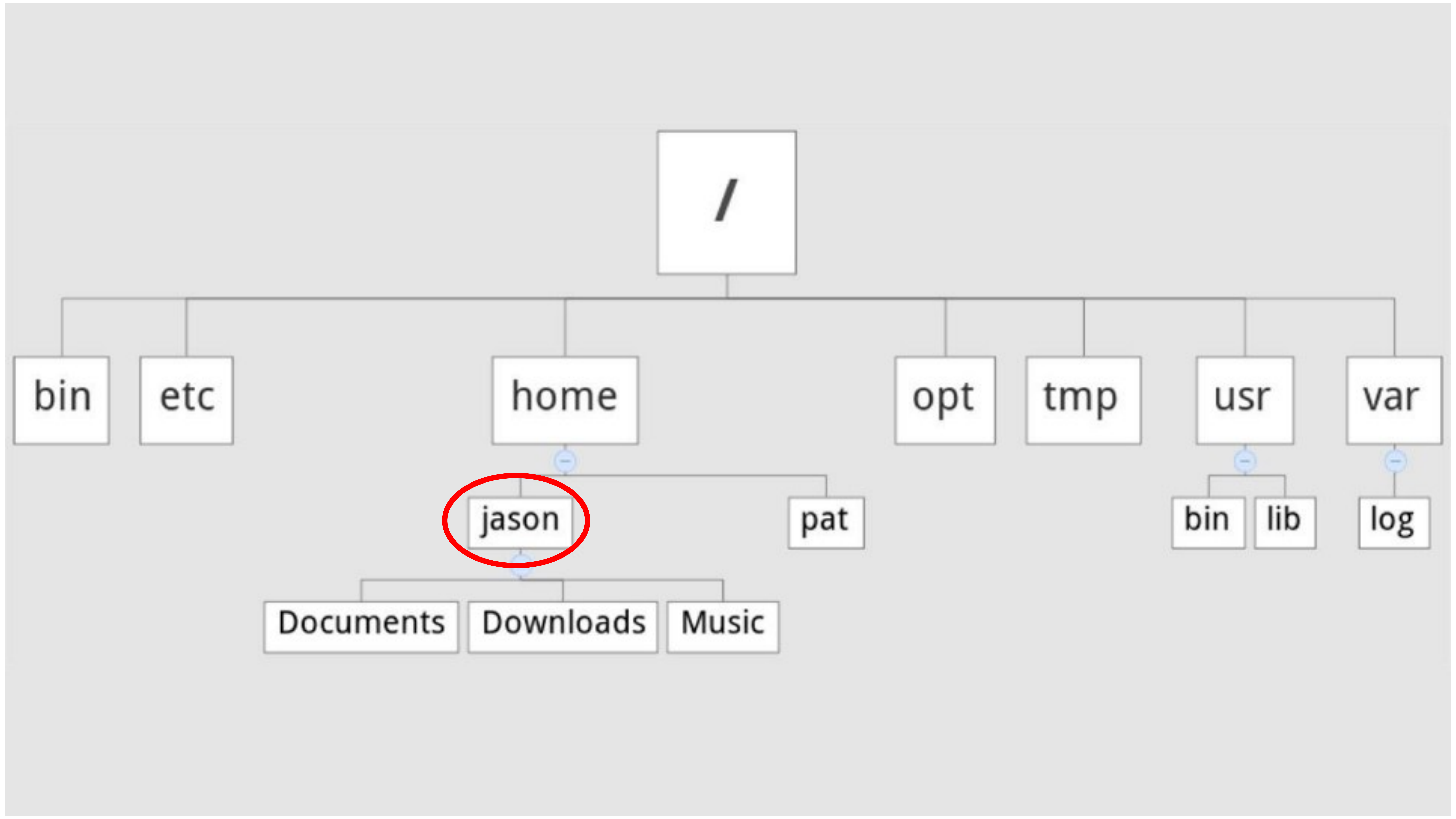
Linux filesystem



Linux filesystem

- When opening a new terminal our current working directory is the user's home.

- Command **`pwd`**:
prints the path of the "working" (current) directory.



Working directory

- The directory where we currently are (shown in the prompt), is called “working directory” or “current directory”.
- By default, the first working directory is the “home” (denoted by the symbol “~”).
- You can see the content of a folder (the list of files and directories) with the command **`ls`** (list).

A simple command: show directory content

- Like all commands in Linux, you can add options to the “`ls`” command to alter its output or influence its behaviour.
- An option is preceded by a dash or a double dash: “`ls -l`” produces a “long format” directory listing; it also shows the permissions, owner, group, size, date and hour of modification.
- “`ls -a`” lists all the files in the directory, including hidden ones (they start with a ‘.’ character).

Man

- Doubts on a command or its options? Use the manual: **`man`**. example:

```
man ls
```

- Displays the manual entry for “`ls`”.

Moving in the filesystem

You can move the current directory using the "`cd`" command (change directory). Note that prompt changes:

- `cd /home`

- you can move "one directory back" with the command

  ```
  cd ..
  ```

- you always return the home directory with

  ```
  cd
  ```

Run **pwd** to see the current working directory (or look at the prompt).

Absolute vs. relative path

- An **absolute** path starts with a "/" (slash) and specifies the entire sequence of directories from the "root" directory (/) up to the specific file/directory being requested:

  ```
  /home/prog2021/prova
  ```

- A **relative** path does not start with a "/" and it is relative to the current directory:

  ```
  cd folder1
  ```

  works **only if** the folder "`folder1`" exists in the current working directory.

Example

Two ways to go in the directory `/var/local`:

- using the absolute path: `cd /var/local`
- using the relative path:

```
cd /

cd var

cd local
```

Create/delete a directory

- You can create a directory with

```
mkdir dir_name
```

- You can delete an **empty** directory with

```
rmdir dir_name
```

- As a safety measure, the directory must be empty before it can be deleted.
- If the path is not absolute, the new directory is created in the current working directory.

Create/delete files

- You can remove files (but not directories) with

  ```
  rm file1 file2 file3
  ```

- you can remove files and directory (recursively) with

  ```
  rm -r file1 file2 file3 dir1 dir2
  ```

- option `-f` forces the operation, skipping warnings.
- Be careful:
  - the files are **DELETED PERMANENTLY** (there is no coming back)
  - with `-rf` you can destroy ALL your data

Exercise

- Create a directory “`test`” in your `home` directory.
- Enter in “`test`” directory and create the “`inside`” directory.
- Return in “`test`” directory and remove “`inside`” directory.
- Remove the “`test`” directory.

Command history and tab completion

- It does not take long before the thought of typing the same command over and over becomes unappealing. One solution is to use the **command line history.**

- By scrolling with the [Up] and [Down] arrow keys, you can find your **previously typed commands**.

- Another time-saving tool is known as **command completion**. If you type part of a file or pathname and then press the [Tab] key, the shell presents you with the remaining portion of the available file/path.

  Example: `cd /h`[Tab]

  presents you a list of completions.

Changing name (or moving a file)?

With the command “`mv`” (move) you can:

- rename a file:

```
mv old_filename new_filename
```

- move a file

```
mv /home/prog2021/filename /path/
```

- move AND rename

```
mv /home/prog2021/old_filename /path/new_filename
```

Warning: if the second file exists (`/path/new_filename` in the example), it will be silently overwritten.

Copying files and directories

With the command “`cp`” you can make a copy of a file or a directory:

```
cp old_name new_name

cp file dir_name/

cp old_name dir_name/new_name

cp -r file1 file2 dir1 dir_out/
```

Warning: if the destination file exists, it will be silently overwritten.

Create/display file content

To create a file:

```
    nano filename
```
(textual interface)

```
    gedit filename
```
(graphical interface on Linux)

To display the contents of the specified file into the screen:

```
    less filename

    gedit filename
```

With less, you can use arrows keys and page up/down keys to navigate up and down.
Hit “q” key to quit.

First and last lines

Show the first 10 and last 10 lines:

```
head filename
tail filename
```

Show the first “n” (e.g., 20) and last “n” lines:

```
head -n 20 filename
tail -n 20 filename
```

Concatenate files

Another way to see file contents is to use the `cat` command:

```
    cat filename
```

This command displays the entire file, so it is not convenient to use it with big files.

It can also be used to concatenate files:

```
    cat file1.txt file2.txt > file3.txt
```

Where “>” redirects the output of `cat` towards the file (instead of standard output), **overwriting its content** (if `file3.txt` already exists, it is overwritten); “>>” appends content instead.

Exercise

- In your home directory, create a new directory called "`exercise`" (`mkdir`)
- Change your directory to the directory `exercise` (`cd`)
- Write your name in the file `name.txt`
- Write your surname in the file `surname.txt`
- Concatenate files `name.txt` and `surname.txt` in the new file `student.txt` (`cat`)
- Visualize the content of the file `student.txt` (`less` or `cat`)

Select lines

To select lines matching a specified “PATTERN” in a file

```
    grep PATTERN filename.txt
```

useful to search text in huge files!

Pipelining

The character “|” allows to use the output of a command as input for another program, e.g.:

```
   grep pippo filename.txt | head
```

returns the first ten lines of the file `filename.txt` that contain “`pippo`”.

Counting lines

To print the number of lines, words, characters:

```
wc filename

cat filename | wc
```

To print only the lines:

```
wc –l filename

cat filename | wc –l
```

“`wc –c`” counts characters instead; “`wc –w`” counts words.

C++ compiler: `g++`

g++

- `g++` is the compiler that we will use for C++. Already installed on Linux.
- It is just a call to `gcc` (compiler for C, which you have already used), specialised for C++. Equivalent to:

```
gcc -xc++ -lstdc++ -shared-libgcc
```

g++

- g++ is the compiler that we will use for C++. Already installed on Linux.
- It is just a call to `gcc` (compiler for C, which you have already used), specialised for C++. Equivalent to:

```
gcc -xc++ -lstdc++ -shared-libgcc
```

- Basic compilation:

```
g++ myprogram.cpp
```

  produces the executable

```
a.out
```

  Option “`-o output`” allows to specify the name of the output:

```
g++ myprogram.cpp -o myprogram
./myprogram
```

g++ — includes

- In C++ you can **include** .cpp/.hpp files with the `include` directive. If you do not specify an **absolute** path, g++ looks for them in the directory containing the .cpp file you are compiling.

- What if the included files are not in the same directory?

- For example you might have a file `"foo.hpp"` that resides in `/home/prog2021/project/include`. To include it in `helloworld.cpp`:

  Write `#include "foo.hpp"` inside `helloworld.cpp` (at the top)

  Compile it with the `–I` option:

  ```
  g++ –I /home/prog2021/project/include/ helloworld.cpp –o helloworld
  ```

- This tells g++ to look for header files specified with `#include` in the directory `/home/prog2021/project/include/`.

g++ — object files

- An object file (usually, with extension `.o`) is the result of the compilation process.
- Object files are source files compiled into binary machine language that may contain (and usually do) references to external functions, defined in other files. Hence, object files have to be “linked” to those references.
- An executable file is the result of linking object files together.

g++ — object files

- An object file (usually, with extension `.o`) is the result of the compilation process.
- Object files are source files compiled into binary machine language that may contain (and usually do) references to external functions, defined in other files. Hence, object files have to be “linked” to those references.
- An executable file is the result of linking object files together.
- Here is the basic idea:
    1. compile each `.cpp` file into a `.o` file using the compiler’s option `-c`;
    2. link the `.o` files (along with libraries, possibly) into an executable.
- Of course, one of these `.cpp` files must define `main()`.

g++ — object files

- Suppose we have `main.cpp`, `foo.cpp` and `bar.cpp` and want to create an executable `fubar`:

```
g++ -c foo.cpp -o foo.o

g++ -c main.cpp -o main.o

g++ -c bar.cpp -o bar.o

g++ foo.o main.o bar.o -o fubar
```

- The first three commands generate `foo.o`, `main.o` and `bar.o` respectively (option `-c`: does not generate executables, only object files). The last one links them together (no option `-c`: generates executable).

g++ — libraries

- **Library:** collection of object files that has been grouped together into a single file.
- Library files are typically named `.lib` and `.dll` (Windows) and `.a` and `.so` (Linux and Mac) for **static** and **dynamic** libraries, respectively.
- A **static library** is copied into the executable file at compile time. A **dynamic library**, instead, exists as a collection of files outside of the executable file. Multiple applications can use a dynamic library without the need for a private copy.
- Using a dynamic library, you only need to recompile the library if you change it but not your application that uses the library. This is not possible with a static library instead: if you make a change, you need to recompile your application since the library will be part of the executable.
- Using a static library makes your code more robust since it does not depend on external references and usually faster at run time since the library’s functions do not need to be called from outside of the executable.

g++ — libraries

- When the linking command encounters a library in its list of object files to link, it looks to see if preceding object files contained calls to functions not yet defined that are defined in one of the library's object files.

- When it finds such a function, it then links in the appropriate object file from the library.

- Libraries are usually linked with an argument of the form

```
-l <library-name>
```

- Example: **`-lm`** links a library containing various mathematical routines (sine, cosine, arctan, square root, etc.).

- Libraries must be listed **after** the object or source files that contain calls to their functions.

g++ — common flags

`-g` turn on debugging. Needed for the debugger/profiler (e.g., gdb and Valgrind — next lectures)

`-Wall` and **`-Wextra`** turn on most warnings

`-O` turn on optimizations (capital “O”, next slide)

`-o` `<name>` name of the output file

`-c` output an object file (.o) rather than an executable file

`-I <include path>` specify an include directory

`-L <library path>` specify where libraries are located

`-l <library>` link with library `lib<library>.a` (esempio: `-lm`)

`-std=c++17` specifies the version of the C++ standard to use.

g++ — optimizations

option	optimization level	execution	code	memory usage	compile time
`-O0`	optimization for compilation time (default)	+	+	-	-
`-O1`	optimization for code size and execution time	-	-	+	+
`-O2`	optimization more for code size and execution time	--		+	++
`-O3`	optimization more for code size and execution time	---		+	+++
`-Os`	optimization for code size		--		++
`-Ofast`	O3 with fast accurate math calculations	---		+	+++

Need help?

Don't tell me you climbed **all the way up here** before trying **Google**??
Pirillo & Fitz
COPYRIGHT 2006 BLAUGH.COM - BRAD FITZPATRICK & CHRIS PIRILLO - SPONSORED BY GODADDY