9 – Exercises on Shell, Compilation & Build Systems

Université Jean Monnet – Saint-Étienne 2026/2027

Quick reference

Command line and paths

Command	Effect
`pwd`	Print the current working directory (absolute path)
`ls`	List files and directories in the current directory
`ls <path>`	List files and directories at `<path>`
`cd <path>`	Change the current directory to `<path>`
`cd ..`	Go up one level (parent directory)
`cd ~`	Go to your home directory

An **absolute path** starts from the root of the filesystem: `/home/alice/music/tracks`.

A **relative path** starts from the current directory: `music/tracks` or `../images`.

Compiling with `clang++`

Command	Effect
`clang++ file.cpp`	Compile `file.cpp`; output binary is `a.out`
`clang++ -o name file.cpp`	Compile and name the binary `name`
`clang++ -o name a.cpp b.cpp`	Compile multiple source files into one binary
`clang++ -I<dir>`	Add `<dir>` to the header search path
`./name`	Run the binary

Linking against libraries

Flag	Meaning
`-L<dir>`	Tell the linker to look for libraries in `<dir>`
`-l<name>`	Link against `lib<name>.so` (or `<name>.dll/lib`, or `<name>.dylib`)
`LD_LIBRARY_PATH` (Linux)	Colon-separated list of directories searched at runtime for
`DYLD_LIBRARY_PATH` (MacOS)	shared libraries
`PATH` (Windows)	

Example: a library `libsndfile.so` sits in `/usr/local/lib` and its headers are in `/usr/local/include`.

```
clang++ -o player main.cpp -I/usr/local/include -L/usr/local/lib -
    lsndfile
```

If the library is a *shared* library (`.so`), the OS must also be able to find it at *runtime*:

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
./player
```

Building with CMake

Command	Effect
`cmake -B <build_dir>`	Configure the project; create `<build_dir>`
`cmake -B <build_dir> -S <src_dir>`	Configure, specifying where `CMakeLists.txt` is
`cmake --build <build_dir>`	Compile and link everything
`cmake --build <build_dir> --target <name>`	Build only the named target

A typical workflow:

```
cd my_project           # CMakeLists.txt is here
cmake -B build          # configure; creates my_project/build/
cmake --build build     # compile; binary appears inside build/
./build/my_binary       # run
```

Problem 1 — Paths, navigation and basic shell commands

1.1 — What does `pwd` print?

The filesystem looks like this. Alice's current directory is `/home/alice/projects/sound`.

```
home/
└── alice
    ├── projects/
    │   ├── sound/
    │   │   ├── main.cpp
    │   │   └── filters
    │   │       └── reverb.cpp
    │   └── images/
    │       ├── main.cpp
    │       └── data
    │           └── photo.png
    └── downloads/
        └── lib.zip
```

After each command sequence below, write what `pwd` would print.

Each row is independent: always start from `/home/alice/projects/sound`.

Commands	`pwd` output
(starting position, no command)	
`cd ..`	
`cd ../images`	
`cd filters`	
`cd ../../downloads`	
`cd /home/alice`	

1.2 — Absolute or relative?

Classify each path as **absolute** or **relative**.

Path	Absolute or relative?
`/home/alice/projects`	
`../images/data`	
`filters/reverb.cpp`	
`/usr/lib`	
`./main.cpp`	

1.3 — Fill in the command

Download this archive and unzip it somewhere on your computer. Open a terminal in the folder `projects`. Write a single shell command that achieves each goal.

Goal	Command
List the contents of `filters/`	
Go to `/home/alice/projects/images/data`	
Go to `/home/alice/projects/images/data` using a relative path	
List the contents of `/home/alice/downloads` without leaving the current directory	
Go to the home directory	

Problem 2 — Compiling with `clang++`

2.1 — One file

You are in `/home/alice/projects/sound` and it contains a single file `main.cpp`.

Write the command to:

Goal	Command
Compile `main.cpp` with the default output name	
Compile it and name the binary `sound`	
Run the resulting binary `sound`	

2.2 — Multiple files and headers

Download this archive and unzip it somewhere on your computer. Open a terminal in the folder `alice/projects/sound`.

`main.cpp` includes `include/filters.hpp` with:

```
#include <filters.hpp>
```

The function `reverb` used in `main` is defined in `filters/reverb.cpp`.

In your terminal, write the commands to:

Goal	Command
Compile both source files into a binary named `sound`	

Problem 3 — Linking against libraries

3.1 — Write the command

Download this archive, unzip it somewhere you can access it. Open a terminal and navigate to the folder `alice/projects/sound`. The project layout is:

```
main.cpp
libs/
    include/
        reverb.h
    lib/
        libreverb.so
        reverb.dylib
        reverb.dll
        reverb.lib
```

In your terminal, write the commands to:

Goal	Command
Compile `main.cpp` into `sound`, with headers and the `reverb` library	
Set your environment so the OS finds the reverb library at runtime	
Run the binary	

3.2 — Short answer

What is the difference between the `-L` flag and `LD_LIBRARY_PATH`? When is each one needed?

Problem 4 — Building with CMake

4.1 — Fill in the command

Download this archive and unzip it somewhere on your computer. Open a terminal in the folder `alice/projects/sound`.

```
CMakeLists.txt
main.cpp
filters/
    reverb.cpp
include/
    reverb.hpp
```

In your terminal, write commands to achieve the following:

Goal	Command
Configure the project into a folder called `build`	
Compile the project	
Rebuild after alterting `filters.cpp`, e.g add a `std::cout` somewhere (same command)	

4.2 — Short answer

Why do we separate the configure step (`cmake -B`) from the build step (`cmake --build`)? In what situation would you re-run configure?