Assignment 3

Université Jean Monnet – Saint-Étienne 2026/2027

Each exercise is independent. Read the full statement before writing any code. Your program must compile and produce exactly the expected output described.

For references on handling `main`'s argument, see the Quick Reference of Lecture 2's exercises on the `main` function.

Exercise 1 — Image tweaker

To complete this exercise, you have to download the archive corresponding to your OS here: windows macos linux. The archive is also available on my website: `https://grybouilli.github.io/introduction-to-programming.html`. Decompress the archive and open the folder it contains in VSCode.

You will write a command-line program that lets the user manipulate images colors.

Invocation:

```
./img_tweaker <input_file> <output_file> <command> [args...]
```

Step 1 — Getting familiar with the provided library

Here is a correspondance table of types you can use (using one is equivalent to writing the other):

Type	Usable Alias
`std::vector<std::vector<std::vector<uint8_t>>>`	`RGBImage`
`std::vector<std::vector<uint8_t>>`	`BWImage`

Look into the file `image_manips.hpp` and read the functions descriptions.

Which function can you use to read an rgb image from a file?

Which function can you use to save an rgb image into a file?

What does `swap_channels` do?

Does `swap_channels` modify the arguments it's given?

What does `convert_rgb_to_bw` do?

Does `convert_rgb_to_bw` modify the arguments it's given?

Step 2 — Writing the main function

The commands

The program will take a minimum of three arguments (`argc == 4`), but may take extra arguments depending on the `command` it is given. You have to implement two commands:

Command name	Extra arguments	Program behavior
`tobw`	*no extra argument*	Converts the image read from `input_file` to black and white and saves the resulting image to `output_file`
`swap`	`<channel1> <channel2>` each one can take any value of `red green blue`	Swaps channels `channel1` and `channel2` of the image read from `input_file` and saves the resulting image to `output_file`. If one of the channel value given as input to the program is neither of `red green blue`, the program should output an error message.

Some invocation examples:

```
./img_tweaker input_file.png output_file.png tobw
./img_tweaker input_file.png output_file.png swap red green
./img_tweaker input_file.png output_file.png swap wrong channel
Error: channels can only take values in {red, green, blue}
```

To do

Write the main function in the file `main.cpp`. Use the functions from `image_manips.hpp` to read, modify and save image files. The program should print the expected usage if the amount of passed arguments is wrong.

Step 3 — Compiling by hand

Your project should have the following layout:

```
src/
└── main.cpp
libs/
├── include/
│   └── image_manips.hpp
└── libimage_manips.so
test/
└── rainbow.png
```

In the terminal, place yourself at the root of the project. Using `clang++`, compile the project. The output program must be named "`img_tweaker`" and placed at the root of the project. You need to link against `libimage_manips.so` (or `libimage_manips.dll` or `libimage_manips.dylib` depending on your OS) in order for the program to have access to the functions from `image_manips.hpp`.

You final command here:

Step 4 — Building with cmake

Create a file `CMakeLists.txt` at the root of the project. Copy and paste the following code inside it:

```
cmake_minimum_required(VERSION 3.14)
project(img_tweaker LANGUAGES CXX C)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_BUILD_TYPE Release)
FILE(GLOB PROJECT_SOURCES src/*.cpp)
add_executable(${PROJECT_NAME} ${PROJECT_SOURCES})
target_include_directories(${PROJECT_NAME} PRIVATE libs/include)
target_link_directories(${PROJECT_NAME} PRIVATE libs)
target_link_libraries(${PROJECT_NAME} PRIVATE image_manips)
```

In the terminal, place yourself at the root of the project. Using `cmake`, generate and build the project in a folder named `build`.

You final commands here:

Step 5 — Testing

Try invoking your compiled programs with the following commands:

```
./img_tweaker test/rainbow.png test/bw_rainbow.png tobw
./build/img_tweaker test/rainbow.png test/grb_rainbow.png swap red green
```

Exercise 2 — Small Synth

To complete this exercise, you have to download the archive corresponding to your OS here: windows macos linux. The archive is also available on my website: https://grybouilli.github.io/introduction-to-programming.html. Decompress the archive and open the folder it contains in VSCode.

This exercise is in two parts. You should try to complete the first part before the second.

Step 1 Generating melodies

The code for this part has to be written in the file `src/create_melody.cpp`.

You will produce a program `create_melody` that creates melodies from a set of triplets passed as arguments. The usage is:

```
./create_melody <note> <octave> <duration> [<note> <octave> <duration>
   ...]
```

There can be an indefinite amount of arguments, as long that amount is multiple of 3 (`(argc-1)%3==0`).

Getting familiar with the synth library

The type `Audio` represents a piece of audio. The `enum class Note` represents the different musical notes.

Give an example usage of the function `create_note`:

Given a variable `Audio audio`, write an example usage of the function `append_note`:

Does the function `append_note` modify its first argument ?

Let's say you have two variables `Audio audio1, audio2;`. Which of `audio1` or `audio2` contains the audio concatenation in the call `concatenate_sounds(audio2, audio1)`?

How would you save a piece of audio to a file? What is the type of that audio file?

Processing the program's inputs

Write your code in the file `src/create_melody.cpp`. It contains a function `parse_note` that takes a `std::string` and returns the corresponding constant from the enum `Note`. You can use it to parse the program arguments.

In the `main` function, if the condition `(argc-1)%3 == 0` is not verified, print the program usage.

Declare three `std::vector` :

```
std::vector<Note> notes;
std::vector<int> octaves;
std::vector<float> durations;
```

Write the code to fill them with the values of the triplets passed as arguments. A little reminder about `argv`:

```
// ./program word 3 4.5
std::string word = std::string(argv[1]);
int three = atoi(argv[2]);
float four_point_five = atof(argv[3]);
```

Hint: you can use a for-loop with an increment of 3, starting from one:

```
for(int arg_idx = 1; arg_idx < argc; arg_idx += 3){
    // argv[arg_idx] -> note
    // argv[arg_idx + 1] -> octave
    // argv[arg_idx + 2] -> duration
}
```

Creating the program's input melody

Using the functions from the `synth` library, write the code to :

- Create a variable `Audio melody`;
- Fill it with the notes provided as arguments to the program;
- Save the melody to a file with the name of your choice, respecting the expected format.

Compiling with `clang++`

Your project should have the following layout:

```
src/
├── main.cpp
libs/
├── include/
│   ├── synth_lib.hpp
│   └── music_files.hpp
├── synth_lib.so
└── music_files.so
```

Open a terminal at the root of the project. Using `clang++`, compile the project. The output program must be named "`create_melody`" and placed at the root of the project. You need to link against `libsynth_lib.so` (or `synth_lib.dll` or `synth_lib.dylib` depending on your OS) in order for the program to have access to the functions from `synth_lib.hpp`.

Building with `cmake`

Create a file `CMakeLists.txt` at the root of the project. Copy and paste the code from Listing

4 inside it. In the terminal, place yourself at the root of the project. Using `cmake`, generate and build the project in a folder named `build`.

You final commands here:

Listing 4: CMakeLists to build create_melody

```
cmake_minimum_required(VERSION 3.14)
project(create_melody LANGUAGES CXX C)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_BUILD_TYPE Release)
FILE(GLOB PROJECT_SOURCES src/create_melody.cpp)
add_executable(${PROJECT_NAME} ${PROJECT_SOURCES})
target_include_directories(${PROJECT_NAME} PRIVATE libs/include)
target_link_directories(${PROJECT_NAME} PRIVATE libs)
target_link_libraries(${PROJECT_NAME} PRIVATE synth_lib)
```

Step 2 Generating melodies from text files

You will write a program that takes the name of a text file and the name of an audio file as input, and write into the audio file the audio described by the content of the text file.

Program invokation:

```
./melody_from_file <input.txt> <output.wav>
```

Meeting another library

The file `music_file.hpp` contains the declaration of a single function `parse_music_file`. This function reads files that contains triplets like the ones described in the previous step, one triplet per line.

Here is an example file: twinkle_little_star.txt

```
C 3 0.5
C 3 0.5
D 3 0.5
D 3 0.5
E 3 0.5
E 3 0.5
D 3 1
```

Let's consider the following code:

```
1 std::vector<Note> notes;
2 std::vector<int> octaves;
```

```
3 std::vector<float> durations;
4
5 parse_music_file("twinkle_little_star.txt", notes, octaves, durations);
6 // ?
```

What do the vectors contain on line 6?

Variable	Content on line 6
`notes`	
`octaves`	
`durations`	

Writing the program

Write your code in `src/melody_from_file.cpp`.

- Print the usage of the program if the number of passed arguments is not 2 —when `argc` is not 3.
- Use the function `parse_music_file` to read the content of `argv[1]` into three `std::vector`.
- Create a melody from the content of those three `std::vector` and save that melody in `argv[2]`. (You can probably copy-paste some code from the previous step).

Compiling with `clang++`

Open a terminal at the root of the project. Using `clang++`, compile the project. The output program must be named "`melody_from_file`" and placed at the root of the project. You need to link against `libsynth_lib.so` (or `synth_lib.dll` or `synth_lib.dylib` depending on your OS) AND `libmusic_files.so` (or `music_files.dll` or `music_files.dylib` depending on your OS).

Building with `cmake` Here is an **incomplete** `CMakeLists.txt` to compile your project:

```
cmake_minimum_required(VERSION 3.14)
project(melody_from_file LANGUAGES CXX C)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_BUILD_TYPE Release)
FILE(GLOB PROJECT_SOURCES src/melody_from_file.cpp)

add_executable(${PROJECT_NAME} ${PROJECT_SOURCES})
target_include_directories(${PROJECT_NAME} PRIVATE libs/include)
target_link_directories(${PROJECT_NAME} PRIVATE libs)
target_link_libraries(${PROJECT_NAME} PRIVATE synth_lib)
```

Here is a description of a few cmake functions:

From these descriptions, try to find what is missing in the above `CMakeLists.txt`, write the missing part and try to build the project with it.

Hint: only a single line is missing.

Testing examples Call your programs with:

Function	Description
`target_include_directories(<project_name> PRIVATE <directory>)`	Equivalent to `clang++ -I <directory>`; indicates where to find header files
`target_link_directories(<project_name> PRIVATE <directory>)`	Equivalent to `clang++ -L <directory>`; indicates where to find folders containing libraries
`target_link_libraries(<project_name> PRIVATE <library>)`	Equivalent to `clang++ -l<library>`; links against `lib<library>.so` (or `<library>.dll/lib` or `<library>.dylib`)

```
./create_melody C 3 0.75 G 3 0.75 Ab 3 0.25 Eb 3 1.25
./melody_from_file tests/mario.txt mario.wav
```