Assignment 5

Université Jean Monnet – Saint-Étienne 2026/2027

Overview

You will write a command-line audio processor that reads a *config file*, then applies a sequence of commands to a sound file to produce a new one. The companion archive provides a ready-to-use `CMakeLists.txt` that fetches and links libsndfile; you do not need to write it. Your work lives entirely in `src/main.cpp`.

Project layout after extracting the archive:

```
assignment/
├── CMakeLists.txt ..... provided — do not modify
├── media/
│   ├── sounds/ ..... provided .wav files
│   │   ├── bassoon.wav
│   │   ├── cello.wav
│   │   ├── clarinet.wav
│   │   ├── flute.wav
│   │   ├── oboe.wav
│   │   ├── saxophone.wav
│   │   ├── trombone.wav
│   │   ├── viola.wav
│   │   └── violin.wav
│   └── configs/ ..... you will create .txt files here
│       ├── config1.txt ..... provided, see below
│       ├── config2.txt ..... provided, see below
│       └── ...
└── src/
    └── main.cpp ..... your file
```

Building

From the project root:

```
cmake -B build
cmake --build build
```

The binary is produced at `build/audio_processor`. Run it as:

```
./build/audio_processor media/configs/config1.txt
```

It takes exactly one argument: the path to a config file. Validate that `argc == 2`; otherwise print a usage message and return `1`.

Config file format

A config file contains a sequence of lines. Blank lines must be ignored. The remaining lines, in order, are:

Line	Meaning
`input <filename>`	Path to the input `.wav` file
`output <filename>`	Path to the output `.wav` file
`concatenate <file1> <file2> ...`	One or more extra files to append after the input
`gain_mult <factor>`	Multiply every sample by `factor` (a floating-point number)

Rules:

- `input` and `output` are mandatory and always appear before any commands.
- Commands are applied **in the order they appear** in the config file.
- `concatenate` takes a variable number of filenames on the same line (at least one).
- `gain_mult` clamps each sample to the range $[-1.0,\ 1.0]$ after multiplication.
- If a line starts with an unrecognised keyword, print a warning to `std::cerr` and skip that line.

Provided `media/configs/example.txt`**:**

```
input media/sounds/bassoon.wav
output media/sounds/ensemble.wav

concatenate media/sounds/cello.wav media/sounds/clarinet.wav media/sounds/trombone.wav
    media/sounds/viola.wav
```

libsndfile reminder

```cpp
#include <sndfile.h>
// Open a file for reading
SF_INFO info;
SNDFILE* snd = sf_open("file.wav", SFM_READ, &info);
if (!snd) { /* handle error */ }
// info.frames   = number of sample frames
// info.channels = number of channels (e.g. 2 for stereo)

// Read all samples into a vector
// Total number of floats = info.frames * info.channels
std::vector<float> samples(info.frames * info.channels);
sf_readf_float(snd, samples.data(), info.frames);
sf_close(snd);
```

```cpp
// Write a vector of samples to a file
SF_INFO out_info = info;      // copy format from input
SNDFILE* out = sf_open("out.wav", SFM_WRITE, &out_info);
sf_writef_float(out, samples.data(), samples.size() / out_info.channels);
sf_close(out);
```

Important: `sf_readf_float` and `sf_writef_float` work with *frames*, not individual samples. For stereo audio, one frame contains two samples (left and right). The total number of floats in the buffer is always `frames * channels`.

Exercise 1 — Parse the config

In `src/parse_configs.cpp`, write a function that reads a config file and populates the following data:

```cpp
std::string              input_path;
std::string              output_path;
std::vector<std::string> commands; // each element is a full command line,
// e.g. "gain_mult 0.5"
//   or "concatenate a.wav b.wav"
```

The function should have the following signature:

```cpp
bool parse_config(const std::string& path,
                  std::string& input_path,
                  std::string& output_path,
                  std::vector<std::string>& commands);
```

Returns `true` on success, `false` if the file cannot be opened or a mandatory field is missing.

Parsing rules:

- Use `std::getline` to read line by line.
- Skip lines that are empty.
- `input_path` and `output_path` should only contain file paths, not “input” or “output”.
- For any other non-blank, non-comment line, store the entire line as a command string (it will be interpreted later).

Verify: print `input_path`, `output_path`, and each command to `std::cout` before moving on to step 2.

Exercise 2 — Load and save audio

In `src/edit_audio.cpp`, write two functions:

```
// Load all samples from a .wav file. Close that file once read.
// Returns an empty vector and prints an error if the file cannot be opened.
std::vector<float> load_wav(const std::string& path, SF_INFO& info);

// Write samples to a .wav file using info as the format template.
// Returns false and prints an error if the file cannot be created.
bool save_wav(const std::string& path,
              const std::vector<float>& samples,
              SF_INFO& info);
```

Verify: load `input_path`, immediately save it to `output_path`, and confirm the output file is a valid `.wav` that sounds identical to the input.

Exercise 3 — Implement the commands

In `src/edit_audio.cpp`, write one function per command.

`gain_mult`

```
void apply_gain_mult(std::vector<float>& samples, float factor);
```

Multiplies every element of `samples` by `factor`, then clamps each result to $[-1.0,\ 1.0]$.

`concatenate`

```
void apply_concatenate(std::vector<float>& samples,
                       const std::vector<std::string>& extra_paths,
                       SF_INFO& info);
```

Loads each file in `extra_paths` in order. For each one, check that the format (sample rate and channel count) matches `info`; if not, print a warning to `std::cerr` and skip that file. Otherwise, append its samples to `samples`. As a reminder, you can use either `insert` or `std::copy` to copy elements from one vector to another:

```
vector1.insert( vector1.end(), vector2.begin(), vector2.end() );
std::copy( vector2.begin(), vector2.end(), std::back_inserter(vector1) );
```

Hint: you can use your existing `load_wav` function here.

Exercise 4 — Dispatch and execute

In `src/parse_configs.cpp`, write a function that takes the list of command strings and applies them in order to the sample buffer:

```
void execute_commands(const std::vector<std::string>& commands,
                      std::vector<float>& samples,
                      SF_INFO& info);
```

You can use the function `split`, defined in `src/parse_configs.cpp`, that returns a `std::vector<std::string>` containing all words seperated by `delimiter` in `s`:

```
std::vector<std::string> split(std::string s, const std::string& delimiter) {
    std::vector<std::string> tokens;
    size_t pos = 0;
    std::string token;
    while ((pos = s.find(delimiter)) != std::string::npos) {
        token = s.substr(0, pos);
        tokens.push_back(token);
        s.erase(0, pos + delimiter.length());
    }
    tokens.push_back(s);

    return tokens;
}
```

For each command string:

1. Read the first token (the keyword).
2. Dispatch with `if` / `else if` / `else`.
3. Parse the remaining tokens for the arguments of that command.
4. Call the corresponding function from step 3.
5. For an unknown keyword, print a warning to `std::cerr`.

If you want to separate the commands arguments from the command itself, you can use either one of the following methods:

```
auto command_tokens = split(commands[i], " ");
// 1. remove first element in command_tokens
command_tokens.erase(command_tokens.begin());
// 2. construct new vector with all elements but the first
auto command_args = std::vector<std::string>(command_tokens.begin()+1, command_tokens.
    end());
```

Exercise 5 — Wire everything together in `main`

`main` should:

1. Validate `argc`.
2. Call `parse_config`; exit with `1` on failure.
3. Call `load_wav` on the input file; exit with `1` if empty.
4. Call `execute_commands`.
5. Call `save_wav` on the output file; exit with `1` on failure.
6. Print `"done"` on success.

Exercise 6 — Testing

Call your program with the given example config files located in `media/configs`. Verify the output files exist and sound as expected.

Exercise 7 — Write your own config files (Optional)

Create at least **two** config files in `media/configs/` that each test a different combination of commands. Use the provided `.wav` files as inputs. Suggested scenarios:

- `quiet.txt`: load `loop.wav`, apply `gain_mult 0.2`, save.
- `long.txt`: load `loop.wav`, concatenate `hit.wav` and `tone.wav`, apply `gain_mult 1.5`, save.

Verify the output files sound as expected.