Assignment 2

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. You can complete each exercise by starting in this session: `https://godbolt.org/z/9vG3od7va` (re-open this link for each exercise). Don't forget to re-export your session link after completing each exercise. You should have a link per exercise.

Exercise 1 — Playlist analyser

You will write a command-line program that analyses a short playlist of songs represented as integer scores (0–100), supplied entirely as arguments to `main`.

Invocation:

```
./playlist 72 95 40 88 55
```

Each argument after the program name is the score of one song.

Step 1 — Argument validation.

If no scores are provided (`argc == 1`), print:

```
usage: ./playlist <score1> <score2> ...
```

and return `1`.

Step 2 — Categorise each score.

Define an `enum class Rating` with three values: `Poor`, `Good`, `Excellent`.

Write a function:

```
Rating rate(int score);
```

that returns:

Condition	Rating
`score < 50`	`Poor`
`50 <= score < 80`	`Good`
otherwise	`Excellent`

Write a second function:

```
void print_rating(Rating r);
```

that uses a `switch` to print `"poor"`, `"good"`, or `"excellent"`.

Step 3 — Loop and report.

In `main`, loop over the arguments (starting at index 1). For each one, convert it with `atoi`, print its score and rating on one line like:

```
score 72: good
```

While looping, track:

- the sum of each score (to compute the average),
- the highest score seen.

Step 4 — Summary.

After the loop, print:

```
average: <integer average>
best: <highest score>
```

Then call `print_rating(rate(best_score))` to print the rating of the best song on a third summary line:

```
best rating: <rating>
```

Expected output for `./playlist 72 95 40 88 55`:

```
score 72: good
score 95: excellent
score 40: poor
score 88: excellent
score 55: good
average: 70
best: 95
best rating: excellent
```

Integer average: truncate, do not round.

Your solution link here:

Exercise 2 — Image colour corrector

You will write a program that applies a correction to the three colour channels of a pixel, with settings supplied on the command line.

Invocation:

```
./correct <r> <g> <b> <offset>
```

`r`, `g`, `b` are the initial pixel channel values (integers); `offset` is an integer added to all three channels.

Step 1 — Argument validation.
Exactly 4 arguments must be provided (so `argc == 5`). Otherwise print:

```
usage: ./correct <r> <g> <b> <offset>
```

and return `1`.

Step 2 — Global pixel state.
Declare three global variables `int r`, `int g`, `int b`, all initialised to `0`.
Write a function:

```
void set_pixel(int rv, int gv, int bv);
```

that assigns the three globals from its arguments.

Step 3 — Clamped offset.
Write two overloads:

```
int apply(const int channel, const int offset); // returns clamped value
void apply(int& result, int channel, int offset); // stores modified channel in result
```

Both add `offset` to `channel` and clamp the result to $[0, 255]$. The first returns the result; the second stores the modified channel in result and returns nothing.

Step 4 — Correction and validity.
In `main`:

1. Read the four arguments and call `set_pixel` to initialise the globals.
2. Print the original pixel values:

   ```
   original: <r> <g> <b>
   ```

3. Call the **reference** overload of `apply` on each global channel with the `offset` given as argument to `main`.
4. Print the corrected pixel values:

   ```
   corrected: <r> <g> <b>
   ```

5. Compute a `bool any_clipped` that is `true` if any channel was clamped during the correction (i.e. the raw value before clamping was outside $[0, 255]$). Print `"clipped"` or `"no clip"`.

Hint for step 5: compute the raw corrected value before clamping and compare it to the valid range.

Expected output for `./correct 200 100 30 80`:

```
original: 200 100 30
corrected: 255 180 110
clipped
```

Expected output for `./correct 100 120 80 20`:

```
original: 100 120 80
corrected: 120 140 100
no clip
```

Your solution link here:

Exercise 3 — Live sound board

You will write a program that simulates a simple sound board with named channels. The program takes a *command* as its first argument and a *value* as its second (when needed).

Invocation:

```
./board <command> [value]
```

Step 1 — Global board state.
Declare two global integers: `int master = 100` and `int monitor = 80`.

Step 2 — Channel enum.
Define:

```
enum class Channel { Master, Monitor };
```

Write a function:

```
Channel parse_channel(std::string name);
```

that returns `Channel::Master` if `name` is `"master"`, `Channel::Monitor` if `name` is `"monitor"`. For any other string, print `"unknown channel"`.

Step 3 — Operations.

Write the following functions:

```
void set_level (int& channel, int value); // assigns value, clamps [0,100]
void boost_level(int& channel, int amount); // adds amount, clamps [0,100]
bool is_hot    (const int& channel);      // true if channel > 90
void print_level(const int& channel);    // prints the value
```

Step 4 — Command dispatch.

`main` accepts the following commands (first argument):

Command	Extra arg	Effect
`set`	value	Set the given channel to value. Then print the new level and whether it is hot.
`boost`	value	Add value to the given channel. Then print the new level and whether it is hot.
`status`	—	Print both channels' levels.

For `set` and `boost`, the channel name is `argv[2]` and the integer value is `argv[3]`. For `status`, no extra arguments are needed.

If the command is not recognised, print `"unknown command"` and return `1`.

Use `if` / `else if` / `else` to dispatch commands (you cannot `switch` on a string).

Expected outputs:

```
./board status
```

```
master: 100
monitor: 80
```

```
./board set master 95
```

```
master: 95
hot
```

```
./board boost monitor 30
```

```
monitor: 100
```

```
hot
```

```
./board boost master -200
```

```
master: 0
not hot
```

Note: each run of the program starts fresh from the initial global values (`master = 100`, `monitor = 80`). There is no persistence between runs.

Your solution link here: