6 – Exercises on `main` and global vs local variables

Université Jean Monnet – Saint-Étienne 2026/2027

Quick reference

Local vs. global variables

```
int global_volume = 80; // global: exists for the entire program,
                        // visible to every function
void print_volume() {
    int local_boost = 10; // local: exists only inside this function
    std::cout << global_volume + local_boost << std::endl;
}   // local_boost is destroyed here; global_volume lives on
```

Key rules:

- A local variable **shadows** a global with the same name inside its scope.
- Modifying a global inside a function affects all subsequent callers.
- Prefer local variables; globals make programs harder to reason about.

Passing arguments to `main`

```
int main(int argc, char* argv[]) {
    // argc : number of arguments (including the program name)
    // argv : sequence of strings; argv[0] is the program name,
    //        argv[1] is the first user-supplied argument, etc.
}
```

Invoked on the command line as:

```
./my_program hello 42
// argc == 3
// argv[0] contains "./my_program"
// argv[1] contains "hello"
// argv[2] contains "42"
```

Reading string arguments

`argv` contains variables of type `char *`, which we don't know how to handle yet.

To manipulate string arguments contained in `argv`, you must first store it in a `std::string` like so:

```
std::string str_arg = argv[1];
```

You can also convert it if you want to do comparison on arguments without having to store them:

```
if(std::string(argv[1]) == "a_string"){
    // do something...
}
```

`atoi` — converting a string argument to an integer

```
#include <cstdlib> // required for atoi

int n = atoi(argv[1]); // converts the string "42" to the int 42
```

`atoi` returns `0` if the string cannot be converted (e.g. `atoi("hello")== 0`). It does *not* signal an error — always make sure `argc` is large enough before reading `argv[i]`.

Part A — Local vs. global variables

Problem 1 — Trace global state

What does this program print?

Global volume

```cpp
#include <iostream>

int volume = 50;

void increase() {
    volume = volume + 10;
}

void reset() {
    volume = 0;
}

int main() {
    std::cout << volume << std::endl; // line A
    increase();
    std::cout << volume << std::endl; // line B
    increase();
    increase();
    std::cout << volume << std::endl; // line C
    reset();
    std::cout << volume << std::endl; // line D
    return 0;
}
```

Line A:

Line B:

Line C:

Line D:

Problem 2 — Shadowing

What does this program print? Pay attention to which `colour` is used at each point.

Shadowing

```cpp
#include <iostream>

int colour = 100;
void paint() {
```

```
    int colour = 200;
    std::cout << colour << std::endl; // line A
}
void tint() {
    colour = colour + 50;
    std::cout << colour << std::endl; // line B
}
int main() {
    std::cout << colour << std::endl; // line C
    paint();
    std::cout << colour << std::endl; // line D
    tint();
    std::cout << colour << std::endl; // line E
    return 0;
}
```

Line A:

Line B:

Line C:

Line D:

Line E:

Problem 3 — Scope lifetime

Does this program compile? If not, identify the line and explain why.

Scope lifetime

```
#include <iostream>

void set_hue(int h) {
    int hue = h * 2;
}

int main() {
    set_hue(45);
    std::cout << hue << std::endl; // line X
    return 0;
}
```

Does it compile?

Why:

Problem 4 — Write the code

Write a program with:

- A global `int` variable `hue` initialised to `0`.
- A function `void shift_hue(int amount)` that adds `amount` to the global `hue`, then clamps it to the range $[0, 360]$ in place.
- A function `void print_hue()` that prints the current value of `hue`.
- A `main` that calls `shift_hue(200)`, `print_hue()`, `shift_hue(250)`, `print_hue()`, `shift_hue(-100)`, `print_hue()`.

Expected output:

```
200
360
260
```

Part B — Arguments to `main`

Problem 1 — Reading `argc` and `argv`

The program below is invoked as:

```
./player Beethoven 120
```

What does it print?

Argument reader

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char* argv[]) {
    std::cout << argc    << std::endl; // line A
    std::cout << argv[0] << std::endl; // line B
    std::cout << argv[1] << std::endl; // line C
    std::cout << argv[2] << std::endl; // line D
    std::cout << atoi(argv[2]) + 10 << std::endl; // line E
    return 0;
}
```

Line A:

Line B:

Line C:

Line D:

Line E:

Problem 2 — Argument count guard

What does this program print for each invocation below?

Argument guard

```cpp
#include <iostream>
#include <cstdlib>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "usage: ./filter <value>" << std::endl;
        return 1;
    }

    int value = atoi(argv[1]);
```

```
11
12      if (value < 0 || value > 255) {
13          std::cout << "invalid" << std::endl;
14          return 1;
15      }
16
17      std::cout << "brightness: " << value << std::endl;
18      return 0;
19  }
```

`./filter` ⇒

`./filter 200` ⇒

`./filter 300` ⇒

`./filter hello` ⇒

What does `atoi("hello")` return, and what does the program therefore print for that call? .

Problem 3 — Spot the bug

This program should print the sum of all integer arguments passed to it. For example, `./sum 10 20 30` should print `60`. It contains bugs.

Buggy argument sum

```
1  #include <iostream>
2  #include <cstdlib>
3  int main(int argc, char* argv[]) {
4      int total = 0;
5      for (int i = 0; i <= argc; i = i + 1) {
6          total = total + atoi(argv[i]);
7      }
8      std::cout << total << std::endl;
9      return 0;
10 }
```

Bug:

Fix:

Hint: what is `argv[0]`, and what does `atoi` return for it? What happens when `i == argc`?

Problem 4 — Write the code

Write a complete program that:

1. Expects exactly two command-line arguments: a tempo (integer) and an artist name (string).

2. If the number of arguments is wrong, prints `"usage: ./concert <bpm> <artist>"` and exits with return value `1`.

3. Otherwise, reads the tempo with `atoi` and prints:
   `"<artist> performing at <bpm> bpm"`
   followed by a tempo label on the next line (`"slow"` below 60, `"medium"` between 60 and 120, `"fast"` between 120 and 180, `"very fast"` otherwise).

Expected output for `./concert 90 Miles`:

```
Miles performing at 90 bpm
medium
```

Part C — Mixing everything

Problem 1 — Command-line image filter

Write a complete program that applies a brightness offset to a fixed set of pixel values.

The program takes one command-line argument: an integer offset (positive or negative). If no argument is given, print a usage message and exit.

The pixels are stored as global variables: `int p1 = 80`, `int p2 = 150`, `int p3 = 30`.

Write:

- A function `void apply_offset(int& pixel, int offset)` that adds `offset` to `pixel` and clamps the result to $[0, 255]$.
- A function `void print_pixels()` that prints `p1`, `p2`, `p3` space-separated on one line.
- A `main` that reads the offset from `argv[1]`, calls `apply_offset` on each pixel, then calls `print_pixels()`.

Expected output for `./filter 100`:

```
180 250 130
```

Expected output for `./filter -50`:

```
30 100 0
```