10 – Exercises on Pointers

Université Jean Monnet – Saint-Étienne 2026/2027

Quick reference

```
int  value = 42;
int* p    = &value;  // p holds the address of value
int  copy = *p;      // dereference: read the value at address p
*p = 100;            // dereference: write through the pointer

int* q = p;          // copying a pointer: q points to the same variable
int* empty = nullptr; // null pointer: points to nothing

if (p != nullptr) { /* safe to dereference */ }
```

Reading a declaration: `int* p` means "`p` is a pointer to an `int`". The `*` belongs to the type, not the variable name.

Part A — Pointers: basics

Problem 1 — What does this print?

Trace the program and fill in each output line.

Listing 2: First steps with pointers

```
#include <iostream>

int main() {
    int volume = 80;
    int* p     = &volume;

    std::cout << volume << std::endl; // line A
    std::cout << *p   << std::endl;  // line B

    *p = 95;
    std::cout << volume << std::endl; // line C
    std::cout << *p   << std::endl;  // line D

    volume = 50;
    std::cout << *p   << std::endl;  // line E

    return 0;
}
```

Line A:

Line B:

Line C:

Line D:

Line E:

Why do lines C and D print the same value?

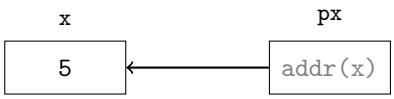
Problem 2 — Draw the memory boxes

After each numbered comment below, draw the state of memory using boxes like the ones shown. Label each box with the variable name, write its value inside, and draw an arrow from each pointer box to the box it points to.

Listing 3: Memory state exercise

```
int brightness = 200;
int contrast  = 40;
int* p        = &brightness;   // (1)
int* q        = p;             // (2)
*q = 150;                      // (3)
p  = &contrast;                // (4)
```

Example format: for `int x = 5; int* px = &x;`:



(1)

(2)

(3)

(4)

Problem 3 — Copying pointers

What does this program print?

Listing 4: Pointer copy

```cpp
#include <iostream>

int main() {
    int track_a = 10;
    int track_b = 20;

    int* p = &track_a;
    int* q = p;        // copy the pointer

    *q = 99;

    std::cout << track_a << std::endl; // line A
    std::cout << track_b << std::endl; // line B
    std::cout << *p    << std::endl; // line C
    std::cout << *q    << std::endl; // line D
    std::cout << (p == q) << std::endl; // line E

    return 0;
}
```

Line A:

Line B:

Line C:

Line D:

Line E:

How many variables does the program have?

Problem 4 — Checking for `nullptr`

Does this program compile and run safely? If not, identify the problem and write a fix.

Listing 5: Null check

```cpp
#include <iostream>

void print_brightness(int* p) {
    std::cout << *p << std::endl;
}

int main() {
    int* p = nullptr;
    print_brightness(p);
    return 0;
}
```

Compiles? **Runs safely?**

Problem:

Fix `print_brightness`, compile and test it.

Problem 5 — Spot the bug

Each snippet has one bug. Identify it and write a corrected version.

Listing 6: Bug hunt

```cpp
// (1) Should print the value of volume through a pointer
int  volume = 75;
int* p     = volume;
std::cout << *p << std::endl;


```

```cpp
// (2) Should change brightness to 255 through a pointer
int  brightness = 100;
int* p          = &brightness;
p = 255;

// (3) Should print the address stored in p
int  x = 42;
int* p = &x;
std::cout << *p << std::endl;
```

Bug 1:

Fix 1:

Bug 2:

Fix 2:

Problem 6 — Write the code

Write a function:

```cpp
void swap(int* a, int* b);
```

that swaps the values of the two integers pointed to by `a` and `b`. Then write a `main` that declares `int left = 30, right = 90`, calls `swap`, and prints both values.

Expected output:

Listing 8:

```
90
30
```

Part B — Pointer arithmetic, arrays, const, references

Problem 1 — Simple pointer arithmetic

Answer the questions below. Assume `int` is 4 bytes wide.

Arithmetic on a pointer

```cpp
int notes[5] = {60, 62, 64, 65, 67};
int* p     = notes;          // p points to notes[0]
```

Expression	Value
`*p`	
`*(p + 1)`	
`*(p + 3)`	
`*(p + 4)`	
`p + 2 == &notes[2]`	

By how many bytes does the address change when you write `p + 1` on an `int*`?

Problem 2 — Trace: looping over an array with a pointer

What does this program print?

Listing 10: Pointer loop

```cpp
#include <iostream>

int main() {
    int pixels[4] = {10, 50, 200, 80};
    int* p      = pixels;
    int* end    = pixels + 4;

    while (p != end) {
        std::cout << *p << std::endl;
        p = p + 1;
    }

    return 0;
}
```

List all printed values in order:

What does `end` point to? Is it safe to dereference?

Problem 3 — `const` pointer vs. pointer to `const`

Fill in the table. For each declaration, answer whether the pointer can be reassigned to a different address, and whether the pointed-to value can be modified through the pointer.

Declaration	Can reassign pointer?	Can modify value?
`int* p`		
`const int* p`		
`int* const p`		
`const int* const p`		

Now classify each line below as **compiles** or **error**, given `int x = 10; int y = 20;`.

Listing 11: Const pointer quiz

```
const int* p = &x;
*p = 99;              // (a)
p  = &y;              // (b)

int* const q = &x;
*q = 99;              // (c)
q  = &y;              // (d)
```

(a)

(b)

(c)

(d)

Problem 4 — References vs. pointers

Fill in the table with **yes** or **no**.

Property	Reference	Pointer
Must be initialised at declaration		
Can be reassigned to refer to a different object		
Can be `nullptr`		
Requires $*$ to access the value		
Can point to / reference an array element		

Then: each function below takes its argument differently. What does the program print, and does the caller's variable change?

Listing 12: Three ways to pass

```cpp
#include <iostream>

void by_value(int v) { v = v + 10; }
void by_ptr (int* p) { *p = *p + 10; }
void by_ref (int& r) { r = r + 10; }

int main() {
    int a = 50, b = 50, c = 50;

    by_value(a);
    by_ptr(&b);
    by_ref(c);

    std::cout << a << std::endl; // line A
    std::cout << b << std::endl; // line B
    std::cout << c << std::endl; // line C

    return 0;
}
```

Line A:	caller's variable changed?
Line B:	caller's variable changed?
Line C:	caller's variable changed?

Problem 5 — Write the code

Write a function:

```cpp
void apply_gain(int* start, int count, int gain);
```

that multiplies every element of the array (given as a pointer to its first element and a count) by `gain`, clamping each result to $[0, 255]$. Use only pointer arithmetic (no index `[]` syntax).

Then write a `main` that declares:

```cpp
int pixels[5] = {30, 80, 120, 200, 50};
```

calls `apply_gain(pixels, 5, 2)`, and prints the result.

Expected output:

Listing 15:

```
60
160
240
255
100
```