UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

INTRODUCTION TO MODERN C++

LECTURE 4

Rémi Géraud

February 11, 2016

École Normale Supérieure de Paris

Lecture 4
Pointers, References, Functions

TABLE OF CONTENTS

1. C++ Memory Model
2. C++ Pointers
3. C++ References
4. Functions

C++ MEMORY MODEL

C++ MEMORY MODEL

When we perform computations, the computer stores our results somewhere

```cpp
int x, y, z;
x = 42;
y = 77;
z = -2;
```

Where? In the computer's memory (RAM). What's memory?

C++ MEMORY MODEL

The memory is a long list of binders called *memory locations*.

C++ MEMORY MODEL

Memory locations are numbered: The zero-th, the first, second etc.

What is the memory location containing the value of *x* ?

```
int x = 42;
std::cout << &x << std::endl;
```

Important note:

- `x` is the *value of x* (= 42)
- `&x` is the *address of x* (= the binder's position).

Note 2: The binder containing *x* is usually quite random.

C++ MEMORY MODEL

The other way around: If you give me an address (= a binder), I can look into it.

```cpp
int x = 42;
std::cout << *(&x) << std::endl;
```

Here I open the binder of *x*. What does it contain?

Important note:

- If `y` is an address (= a binder position = ”pointer”)
- Then `*y` is a value (= the contents of the binder)

C++ MEMORY MODEL

Small exercise:

```
int x = 42;
int y = 73;
std::cout << *(&x + 1) << std::endl;
```

What happens? Why?

C++ POINTERS

POINTERS IN C++

In C++, a pointer type is defined by adding a star symbol:

```
int  x = 42;    // x has type ”integer” and value 42
int* y = &x;    // y has type ”pointer to integer”
```

If you follow, `*y = 42`.

Pay very close attention with all these * and & floating around!

C++ POINTERS: WHY?

Why do we use pointers? A typical scenario is as follows:

- You can put a lot of stuff in a binder.
- Instead of moving everything around, making copies,
- You just say “look in binder 4372”.

Less copies = Faster code

Note: We’ll meet a lot the “null pointer”, `nullptr`.

C++ POINTERS: A WORD OF CAUTION FOR C PROGRAMMERS

We don't use C++ pointers the way we use C pointers.

- In fact we try to avoid using them as much as possible
- Abuse of pointers leads to dangerous, hard-to-debug and hard-to-optimize code
- It is almost always possible to to *without* pointers…
- … at least *raw* pointers.

C++ REFERENCES

C++ REFERENCES

Less powerful than pointers, but often useful, are *references*.

A reference is just “another name” for a variable.

C++ REFERENCES

Example:

```
int  a = 42;
int& b = a;    // Create alias b of a
b = 73;
std::cout << a << std::endl;
```

This program prints 73, because a and b are the same thing.

Pay very close attention with all these * and & floating around!

POINTERS AND REFERENCES: SHORT SUMMARY

Remember this:

`int x;`	Declaration of a variable x
`&x`	“Address of” x = Pointer to x
`*y`	“Contents of” binder at address y (dereference)
`int* y = &x;`	$y =$ address of x = pointer to x
`int& y = x;`	x and y are forever the same thing

Of course the same applies with other types (`float`, etc.).

You must know these by heart.

There will be questions during the midterm

FUNCTIONS

C++ FUNCTIONS

You already met functions in the homework and lab sessions.

A function looks like this:

```
double myFunction(float a, float b, float c) {
    double x;
    // Do some stuff
    return x;
}
```

Some vocabulary:

- This is a *function declaration*
- `a`, `b`, and `c` are called *arguments*
- `x` is the *return value* of `myFunction`.
- `myFunction` has *return type* `double`

Note: What is the type of `myFunction`?

C++ FUNCTIONS

To use this function,

```cpp
double myFunction(float a, float b, float c) {
    double x;
    // Do some stuff
    return x;
}
```

we use the following notation:

```cpp
myFunction(3, 4, 5);
```

This is a *function call*. Example: `double x = cos(42);`

C++ FUNCTIONS

Remark: You can sometimes use *type inference* (keyword `auto`):

```cpp
auto mymax() {
    return 3.14;  // mymax will return float
}
```

Beware: Type inference in C++ is not perfect!

C++ FUNCTIONS AND PURITY

C++ functions makes it easier to reuse and organise code.

They are basic “building blocks” of programs.

Note: A function is *pure* when it gives the same output every time it is called with the same input.

Whenever possible, be pure

It makes your programs more robust and easy to debug

IMPURE C++ FUNCTION

```cpp
#include <iostream>

int main() {
  int x = 0;
  myfunction(x);
  myfunction(x);
}

void myfunction(int& y) {
  y = y + 1;
  std::cout << y << std::endl;
}
```

QUESTIONS?

Lab Session
Headers, Linked Lists, Recursion, and Dynamic programming