# Cpp Introduction

André Cerqueira

FCUP

2024



REPÚBLICA PORTUGUESA | CIÊNCIA, TECNOLOGIA E ENSINO SUPERIOR


Fundação para a Ciência e a Tecnologia

Este trabalho foi financiado por: Projeto 10110190 – EUROCC2, com apoio financeiro da FCT/MCTES através de fundos nacionais (PIDDAC).

# Structure of a C++ Program


```cpp
#include <iostream>

void greet() {
    std::cout << "Welcome to SYCL
    ↪  Learning!" << std::endl;
}

int main() {
    std::cout << "Hello, C++ World!"
    ↪  << std::endl;
    greet();
    return 0;
}
```

- **Comments**:
  - Can be added anywhere in the code
  - Single-line: `//`
  - Multi-line: `/* */`
- **Compiler Directives**
  - `#include` tells the compiler to include libraries.
- **main() Function**
  - Starting point of execution for the program
  - Statements end with a semicolon (`;`)
  - Typically ends with `return 0;` to indicate successful execution

# Fundamental Data Types


- In C++, constants and variables have specific types, which define the kind of data they can hold and the range of values they represent.

| | |
|---|---|
| char | A character or small integer, typically used to store ASCII characters. Range: $-128$ to 127 |
| int | Standard integer type. Range: $-2^{31}$ to $2^{31} - 1$ |
| float | Single-precision floating-point number. Precision: 7 digits |
| double | Double-precision floating-point number. Precision: 15 digits |
| bool | Boolean type, represents true or false |

- Larger or specialized types are available, like ‘short int‘ and ‘long int‘, to represent different ranges of values.

# Arithmetic Operators


- Basic arithmetic operators are used for both integer and floating-point types:

  $+$ Addition: Adds values, e.g., $2 + 5$

  $-$ Subtraction: Subtracts values, e.g., $41 - 32$

  $*$ Multiplication: Multiplies values, e.g., $4.23 * 3.1e - 2$

  / Division: Divides values. Integer division drops the remainder (e.g., $10/3 = 3$), while floating-point division gives a precise result (e.g., $10.0/3 = 3.333...$)

  % Modulus: Finds the remainder after division, only for integers (e.g., $17\%5 = 2$)

- Note: Division behaves differently for integer vs. floating-point types.

# C++ Input and Output


- **Include** <iostream> for input and output functions (cout and cin).
- **Output (cout)**
  - Use cout with << to print values; add << between different values or types.
  - cout does not add spaces automatically: add them as needed.
- **Input (cin)**
  - Use cin with >> to read values into variables from the user.
  - cin skips leading whitespace and stops at trailing whitespace.

**Example Output (cout):**

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
  int x = 5;  char c = 'Y';  double
  ↪  y = 4.5;
  cout << "Hello world" << endl;
  cout << " c = " << c << "\ny is "
  ↪  << y << endl;
  return 0;
}
```

**Example Input (cin):**

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
  int x;  char c;  double y;
  cout << "Enter an integer and
  ↪  double separated by spaces: "
  ↪  << endl;
  cin >> x >> c >> y;
  return 0;
}
```

# Understanding Pointers


- **Pointers** are variables that store memory addresses.
- Each pointer has a **type** (e.g., `char`, `int`, `double`), which determines the size of data it points to.
- A pointer's type affects how far it moves in memory when incremented.

# Pointer Arithmetic


## Pointer Arithmetic Operations

- p++: Move to the next memory location for the pointer's type.
- *p: Access the value at the current address.
- *p++: Increment the pointer, then access the new location.
- (*p)++: Access the value, then increment the value itself.

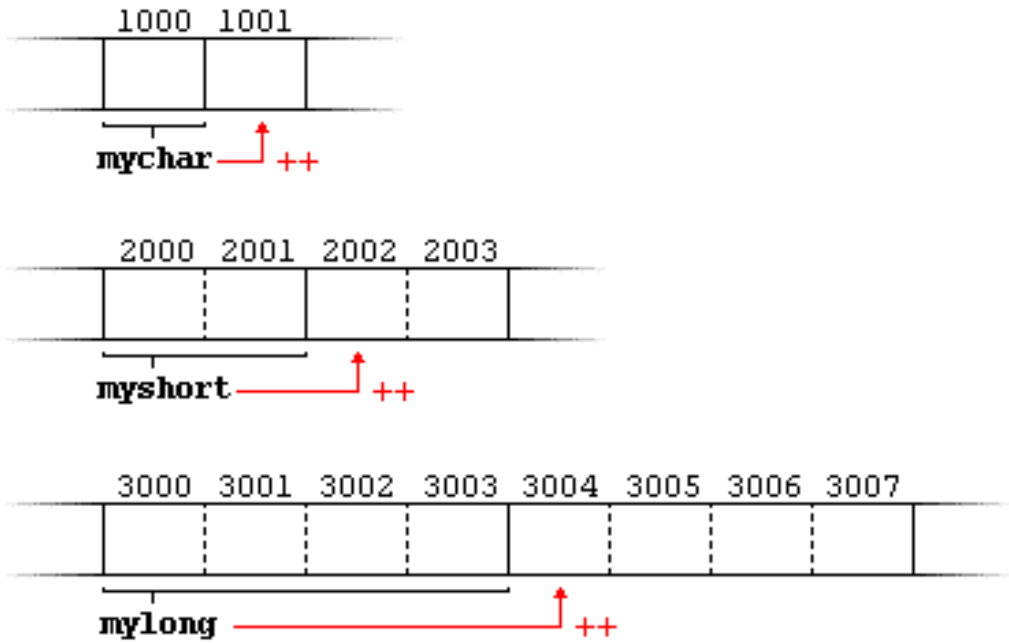


Figure: Memory blocks for different pointer types

# Arrays vs. `std::vector`


**C-style Array**

- Fixed-size, defined at compile-time.
- Memory is contiguous, but size cannot be changed.
- Syntax: `int arr[5];`
- Faster access but lacks flexibility.

**std::vector**

- Dynamic size, can be resized at runtime.
- Automatically manages memory.
- Syntax: `std::vector<int> vec;`
- Provides flexible methods (e.g., `.push_back()`).

# C-style Array Example


**C-style Array Code:**

- Fixed-size array.
- Demonstrates declaration, initialization, and accessing elements.

```cpp
#include <iostream>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    for (int i = 0; i < 5; ++i) {
        std::cout << "Element " << i << ": " << arr[i] << std::endl;
    }

    return 0;
}
```

# std::vector Example


**std::vector Code:**

- Dynamic-size vector.
- Demonstrates adding elements and accessing them.

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec;

    vec.push_back(1);
    vec.push_back(2);
    vec.push_back(3);
    vec.push_back(4);
    vec.push_back(5);

    for (int i = 0; i < vec.size(); ++i) {
        std::cout << "Element " << i << ": " << vec[i] << std::endl;
    }

    return 0;
}
```

# Lambda Expressions in C++


## What are Lambda Expressions in C++?

- **Anonymous Functions**: Inline, unnamed functions that can be defined directly within code.
- **Inspired by Lambda Calculus**: Implement function abstraction, allowing the definition of functions as expressions.
- **Usage in C++**: Useful for short, temporary functions, often passed as parameters in algorithms (e.g., ‘std::sort‘, ‘std::for_each‘).

## Advantages of Lambda Expressions

- Enable cleaner, more concise code, especially for functional programming patterns.
- Allow capturing variables from the surrounding scope for flexible usage.
- Useful in parallel computing contexts, such as SYCL, for defining operations inline.

# Syntax of Lambda Expressions in C++


**Lambda Expression Syntax:**

- General syntax: `[capture](parameters) { body };`
- Example: `auto add = [](int x, int y) { return x + y; };`
- **Capture**: Specifies variables from the surrounding scope.

# Examples of Lambda Expressions in C++


**Examples of C++ Lambda Expressions:**

## 1. Basic Lambda

```
[]() { std::cout << "Hello, World!"; };
```

## 2. Lambda with Parameters

```
[](int x) { return x * x; };
```

## 3. Using Capture

```
[a](int x) { return x + a; };
```

# Memory Management in C++


## Memory Management in C++

- **Static vs. Dynamic Memory**:
  - *Static Memory*: Allocated at compile-time (e.g., arrays with fixed size).
  - *Dynamic Memory*: Allocated at runtime using `new` and `delete`.
- **Dynamic Allocation**:
  - Use `new` to allocate memory and `delete` to free it.
  - Example: `int* ptr = new int[10];` `delete[] ptr;`
- **Smart Pointers (C++11)**:
  - Automatically manage memory and prevent memory leaks.
  - Types: `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`.

# Importance of Memory Management


**Why is Memory Management Important?**

- Essential for performance-critical applications, like those in SYCL.
- Prevents memory leaks and undefined behavior by properly managing resources.
- Smart pointers reduce the need for manual memory management, enhancing code safety and robustness.
- Efficient memory usage directly impacts performance, especially in parallel and high-performance computing.

# Memory Management in C++: Code Example


## Code Example: Dynamic Allocation and Smart Pointers

```
int* ptr = new int[5];
for (int i = 0; i < 5; ++i) {
    ptr[i] = i * 2;
}
delete[] ptr;

#include <memory>

std::unique_ptr<int[]> uniquePtr(new int[5]);
for (int i = 0; i < 5; ++i) {
    uniquePtr[i] = i * 3;
}

std::shared_ptr<int> sharedPtr = std::make_shared<int>(10);
std::weak_ptr<int> weakPtr = sharedPtr;
```


# Thank You!

Questions?