Modern Programming Principles & Practice

Feb-May 2025

sruthi.padathara@dorset.ie

Why use a pointer (int* data;) in C++?

The * (asterisk) means that data does not store an actual integer value but rather **stores the memory address** of an integer.

A **pointer** is a variable that stores the memory address of another variable.
It allows **direct memory access and manipulation**, making operations like **dynamic memory allocation** and **efficient data structures** possible.

Dynamic Memory Allocation:

- A pointer allows you to allocate memory dynamically using new, instead of using fixed memory allocation (stack memory).
- This is useful when the size of the data is not known at compile time.

Deep Copy in Copy Constructors:

- If we store data in a normal integer (int data;), the default copy constructor performs a **shallow copy**, meaning both objects will share the same memory.
- Using a pointer allows us to create a **deep copy**, meaning each object gets its own independent copy of the data.

Efficient Memory Management:

- Objects with dynamically allocated memory can be **deleted** when they are no longer needed, avoiding memory waste.

Example: Pointer in Action

```
#include <iostream>
using namespace std;

int main() {
    int a = 10;
    int* ptr = &a;  // Pointer stores the address of 'a'

    cout << "Value of a: " << a << endl;         // 10
    cout << "Address of a: " << &a << endl;      // Some memory address
    cout << "Value stored in ptr: " << ptr << endl;  // Address of 'a'
    cout << "Value at ptr (dereferencing): " << *ptr << endl; // 10

    return 0;
}
```

ptr stores the address of a
***ptr (dereferencing) gives the value stored at that address**

Why Do We Need a Copy Constructor in C++?

A **copy constructor** is used to create a new object by copying an existing object's values. It is essential for **deep copying** objects that contain dynamically allocated memory.

Key Reasons to Use a Copy Constructor

1. Copying Objects

- When passing an object by **value** to a function.
- When returning an object from a function.

2. Preventing Shallow Copy Issues

- The default copy constructor performs a **shallow copy** (only copies memory addresses).
- If the object has **dynamic memory**, a shallow copy can lead to issues like **double deletion**.

3. Preserving Object Integrity

- Ensures that each object has its own copy of resources, avoiding unexpected modifications.

4. Used in Container Classes

- Copy constructors are essential when storing objects in **STL containers** like vector, map, etc.

Move Constructor in C++11

A **move constructor** is a special constructor in C++11 that transfers ownership of resources **instead of copying**. It improves performance by avoiding **deep copies** when objects are temporary.

Why &&obj in the Move Constructor?

Syntax

ClassName(ClassName&& obj)

uses **double ampersand (&&)**, which means **rvalue reference**.

- ClassName&& obj accepts **temporary (rvalue) objects**, allowing the constructor to **steal (move)** their resources instead of copying them.
- It prevents unnecessary deep copies and speeds up performance.

Copy Constructor vs. Move Constructor in C++

Copy Constructor (Deep Copy)

- Used when an object is **passed by value** or explicitly copied.
- Performs **deep copy** (allocates new memory and duplicates data).
- Can be **expensive** if the object has dynamically allocated memory.

Move Constructor (Efficient Transfer)

- Introduced in **C++11**.
- Used when an object is **moved** (e.g., passing temporary values).
- Transfers **ownership of resources** instead of copying.
- Avoids **unnecessary memory allocations**, improving efficiency.

Feature	Copy Constructor	Move Constructor
Purpose	Creates a deep copy	Transfers ownership
Parameter	const Class& obj	Class&& obj (rvalue reference)
Memory Usage	Allocates new memory	No new memory allocation
Efficiency	Slow (extra allocation & copying)	Fast (just pointer swap)
When Used	Passing objects by value	Moving temporary objects