1. Basic Constructor Questions

✅ What is a constructor in C++?

A **constructor** is a special member function of a class that is automatically called when an object is created. It initializes the object's data members.

cpp
CopyEdit

```cpp
class Example {
public:
    int x;
    Example() { x = 10; } // Constructor
};
```

Key Points:

- Same name as the class.
- No return type (not even `void`).
- Called automatically when an object is created.

✅ Types of constructors?

1. **Default Constructor:** No parameters.
2. **Parameterized Constructor:** Accepts parameters.
3. **Copy Constructor:** Initializes an object from another object.
4. **Move Constructor:** Transfers ownership of resources.
5. **Destructor:** Cleans up resources.

cpp
CopyEdit

```cpp
class Demo {
public:
    int x;

    Demo() { x = 0; }  // Default Constructor
    Demo(int val) { x = val; }  // Parameterized Constructor
};
```

✅ Can a constructor be virtual?

🚫 **No.** A constructor cannot be `virtual`.
Reason:

- Virtual functions rely on the **vtable**, which is set up **after** the constructor runs.
- A constructor is responsible for initializing the vtable, so it can't be virtual.

💡 **Alternative:** Use a **virtual destructor** for polymorphism.

cpp
CopyEdit

```
class Base {
public:
    virtual ~Base() {} // Allows proper cleanup in derived classes
};
```

✅ Why do we need a user-defined constructor when C++ provides a default one?

The compiler provides a **default constructor** only when **no constructor is defined**.
If we want to:

- **Initialize members with specific values.**
- **Allocate dynamic memory.**
- **Enforce certain constraints on object creation.**

Then, we define a **user-defined constructor**.

✅ What happens if a constructor is private?

If a constructor is private, objects **cannot be created outside the class**.

✔ **Use case: Singleton pattern** (Restricting multiple instances of a class).

cpp
CopyEdit

```
class Singleton {
private:
```

```
    Singleton() {}  // Private constructor

public:
    static Singleton& getInstance() {
        static Singleton instance; // Only one instance
        return instance;
    }
};
```

---

2. Copy Constructor & Copy Assignment Operator

✅ What is a copy constructor?

A **copy constructor** creates a new object by copying an existing object.

cpp
CopyEdit

```
class Example {
public:
    int x;
    Example(int val) { x = val; }
    Example(const Example& obj) { x = obj.x; } // Copy Constructor
};
```

✅ Difference between copy constructor and copy assignment?

Feature	Copy Constructor	Copy Assignment Operator
Purpose	Creates a new object from an existing one	Assigns an existing object to another
Called When?	Object is initialized	Object is already created and assigned a new value
Syntax	`ClassName (const ClassName&)`	`ClassName& operator=(const ClassName&)`

cpp
CopyEdit

```
Example e1(10);
Example e2 = e1; // Copy Constructor
e2 = e1; // Copy Assignment
```

✅ When is the copy constructor called?

1. When **passing an object by value**.
2. When **returning an object by value**.
3. When explicitly invoking it (`Class obj2(obj1);`).

✅ What happens if we don't define a copy constructor?

The compiler generates a **default copy constructor** that performs a **shallow copy** (bitwise copy of members).

💡 **Problem:** If a class has **dynamic memory**, the default copy constructor will result in **double deletion**.

cpp
CopyEdit

```
class Demo {
    int* ptr;
public:
    Demo(int val) { ptr = new int(val); }
    ~Demo() { delete ptr; }
};
```

✅ Why should we use deep copy instead of shallow copy?

A **shallow copy** only copies the pointer, not the actual data.
A **deep copy** creates a new copy of the resource.

cpp
CopyEdit

```
class DeepCopy {
    int* data;
public:
    DeepCopy(int val) { data = new int(val); }
```

```
    DeepCopy(const DeepCopy& obj) { data = new int(*obj.data); }  // Deep Copy
};
```

✅ Explain Rule of Three?

If a class **manages dynamic memory**, you should define:

1. **Destructor** (`~ClassName()`)
2. **Copy Constructor** (`ClassName(const ClassName&)`)
3. **Copy Assignment Operator** (`ClassName& operator=(const ClassName&)`)

Otherwise, the default shallow copy can lead to memory issues.

---

3. Move Constructor & Move Assignment

✅ What is a move constructor?

A **move constructor** transfers ownership of resources instead of copying.

cpp
CopyEdit

```
class MoveExample {
    int* data;
public:
    MoveExample(int val) { data = new int(val); }

    // Move Constructor
    MoveExample(MoveExample&& obj) noexcept {
        data = obj.data;
        obj.data = nullptr; // Nullify the source
    }
};
```

✅ Why do we need move semantics?

- Avoids **expensive deep copies**.

- Improves **performance** when dealing with large objects.

Example:

cpp
CopyEdit
```cpp
std::vector<std::string> v;
v.push_back("hello");  // Moves instead of copying
```

✅ Explain Rule of Five?

If a class has **dynamic memory**, define these five:

1. **Destructor**
2. **Copy Constructor**
3. **Copy Assignment Operator**
4. **Move Constructor**
5. **Move Assignment Operator**

---

4. Constructor Behavior & Edge Cases

✅ Can a constructor be `explicit`?

Yes! It prevents **implicit conversions**.

cpp
CopyEdit
```cpp
class Example {
public:
    explicit Example(int x) {}  // No implicit conversion
};

Example e = 10; // ❌ Error (implicit conversion blocked)
```

✅ What is constructor delegation?

Calling one constructor from another.

cpp
CopyEdit

```
class Example {
    int x;
public:
    Example() : Example(10) {} // Delegating
    Example(int val) { x = val; }
};
```

✅ What is member initializer list and why use it?

✔ Faster than assigning values in the constructor body
✔ Mandatory for `const` and reference members

cpp
CopyEdit

```
class Example {
    const int x;
public:
    Example(int val) : x(val) {}  // Member initializer list
};
```

---

5. Advanced Topics

✅ What is Return Value Optimization (RVO)?

The compiler **eliminates unnecessary copies** when returning objects.

cpp
CopyEdit

```
Example createObject() {
    return Example(); // RVO eliminates copy/move
}
```

✅ Why is `std::move` needed?

To **convert an lvalue to an rvalue**, allowing move semantics.

cpp
CopyEdit

```
std::string s = "hello";
std::string s2 = std::move(s);  // Moves instead of copying
```

✅ What is the Rule of Zero?

If possible, **avoid manual resource management** by using **smart pointers** (`std::unique_ptr`, `std::shared_ptr`).