❤️‍🔥

Object Oriented programming

Tags	lecture
Date	@February 22, 2025
Revised	10
Status	Mastered

Although, we can do the normal programming in C++ but in order to apply the real world problems and to solve them,

we need to use the object oriented programming.

OOPs is generally the better way to write the code.

Classes & Objects

- Objects are entities in real world.
- Classes is like the blue print of these objects (Group of objects)

> Methods are the functions / member functions of class because they are the members of class.

Access Modifiers

These are the keywords which are used to set the accessibility of members (attributes / methods) of a class. There are of three types -

1. **Private** - data & methods accessible inside class
2. **Public** - data & methods accessible to everyone

3. **Protected** - data & methods accessible in class and in to the derived class (Inheritance concept).

By default, all the members of class are private. So we really need to specify the access modifiers.

Depending on the situation,

that which data should have the access outside the class and which doesn't , we categorize them into the *public and private* access modifiers.

In order to give the access of protected data to the main function or any other class, we take the help of getter and setter methods.

Remember, we can flag the *getter and setter* methods as public to access the protected data inside the main function or maybe in any of the class.

Example

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

class Teacher
{
private:
    double salary;

public:
    string name;
    string dept;
    string subject;

    void display()
    {
        cout << "Name = " << name << endl;
        cout << "Dept = " << dept << endl;
        cout << "Subject = " << subject << endl;
        cout << "Salary = " << salary << endl;
```

```
    }

    // setter method
    void setter(int s)
    {
        salary = s;
    }
};

int main()
{
    Teacher t1;
    t1.name = "Steve";
    t1.dept = "Computer";
    t1.subject = "Programming";
    t1.setter(30000);

    t1.display();
    return 0;
}
```

Setter Methods

Purpose: Setter methods are used to **set** the value of a private or protected attribute. They allow controlled modification of the attribute's value, and can include validation logic to ensure the values being assigned are valid.

Properties:

- **Access Level**: Public
- **Return Type**: Typically `void`
- **Parameters**: Takes a parameter of the same type as the attribute to be set
- **Naming Convention**: Usually prefixed with `set` followed by the attribute name

Example:

```
class Teacher
{
private:
    double salary;

public:
    void setSalary(double sal)
    {
        if (sal >= 0) // Validation to ensure salary is non-negative
        {
            salary = sal;
        }
    }
};
```

Getter Methods

Purpose: Getter methods are used to retrieve the value of a private or protected attribute. They provide read-only access to the attribute's value.

Properties:

- **Access Level**: Public
- **Return Type**: Same as the attribute's type
- **Parameters**: None
- **Naming Convention**: Usually prefixed with `get` followed by the attribute name

Example:

```
class Teacher
{
private:
    double salary;

public:
    double getSalary()
```

```
    {
        return salary;
    }
};
```

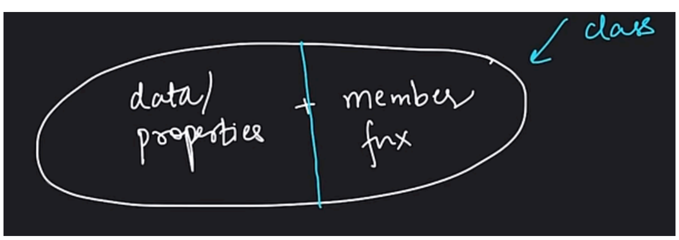
Rules and Best Practices

1. **Encapsulation**: Always use setter and getter methods to access and modify private or protected attributes. This ensures that the internal state of an object is not exposed or altered directly.
2. **Validation**: Implement validation logic in setter methods to ensure that only valid values are assigned to attributes.
3. **Read-Only Attributes:** Use getter methods to provide read-only access to attributes. If an attribute should not be modified after initial assignment, only provide a getter method.
4. **Consistency**: Follow naming conventions to make the code more readable and maintainable.
5. **Documentation**: Document setter and getter methods to clearly explain their purpose and any validation rules applied.

---

4 main pillar of OOP

- **Encapsulation** - wrapping up of data & member functions in a single unit called class.



Encapsulation is useful when there is need of data hiding, for eg- In bank, it might be very necessary to hide the balance of any account.

I mean, when we are working on the big projects like bank accounts, there might be many classes in many different files. So it is necessary to use the private, public and protected access modifiers on those situations.

For eg-

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

class Account{
private:
    double balance; // Private or protected attribute
public:
    string userid;
    string username;
};
```

Constructors in C++

What is a Constructor?

A **constructor** is a special member function of a class that is automatically called when an object of the class is created. It is primarily used to **initialize objects.**

Key Characteristics of a Constructor:

- It has the **same name** as the class.
- It **does not** have a return type (not even `void`).
- It is **automatically invoked** when an object is instantiated.
- It can be **overloaded**, meaning multiple constructors with different parameter lists can exist & it is the example of **Polymorphism**.
- If no constructor is provided, C++ provides a **default constructor** that does nothing.

Types of Constructors in C++

1. Default Constructor (No parameters)

A constructor that takes no parameters and initializes an object with default values.

Example:

```
#include<iostream>
#include<string>
using namespace std;
class Teacher{
public:
    string name;
    string dept;
    string subject;
    double salary;

    Teacher(){
        dept = "Science";
    }

    void display(){
        cout <<"Name: " << name << endl;
        cout <<"Department: " << dept << endl;
        cout <<"Subject: " << subject << endl;
    }

};

int main(){
    Teacher t1;
    t1.name = "Dharamveer";
    //t1.dept = "Applied mathematics";
    t1.subject = "Maths";
    t1.salary = 45000;
```

```
    t1.display();
    return 0;
}
```

Output:

```
Name: Dharamveer
Department: Science
Subject: Maths
```

2. Parameterized Constructor

A constructor that takes arguments to initialize an object with specific values.

Example:

```
#include<iostream>
#include<string>
using namespace std;
class Teacher{
public:
    string name;
    string dept;
    string subject;
    double salary;

    Teacher( string n, string d, string s, double sal){
            name = n;
            dept = d;
            subject = s;
            salary = sal;
    }

    void display(){
        cout <<"Name: " << name << endl;
```

```
        cout <<"Department: " << dept << endl;
        cout <<"Subject: " << subject << endl;
    }

};

int main(){
    Teacher t1("Dharamveer", "Science", "Mathematics", 45000 );

    t1.display();
    return 0;
}
```

Output:

```
Name: Dharamveer
Department: Science
Subject: Mathematics
```

3. Copy Constructor

A constructor that creates a new object as a copy of an existing object.

Example:

```
// This code is the example of the default copy constructor
#include<iostream>
#include<string>
using namespace std;

class Teacher {
public:
    // Attributes initialization
    string name;
    string dept;
    string subject;
```

```
    double salary;

    // Parameterized constructor
    Teacher(string name, string dept, string subject, double salary){
    this→name = name;
    this→dept = dept;
    this→subject = subject;
    this→salary = salary;
    }

    //Output method
    void getInfo(){
        cout << "Name: " << name << endl;
        cout << "Department: " << dept << endl;
        cout << "Subject: " << subject << endl;
        cout << "Salary: " << salary << endl;
    }
};

int main(){
    Teacher t1("Shradha", "Computer Science", "C++", 25000);
    Teacher t2(t1);

    t2.getInfo();
    return 0;
}
```

Output:

```
Name: ShradhaD
epartment: Computer Science
Subject: C++
Salary: 25000
```

Manual copy constructor CORE CONCEPT

A **copy constructor** is a special constructor that initializes an object using another object of the same class. It is used to:

1. Create a copy of an existing object.
2. Perform a **deep copy** (if necessary) to ensure that the new object has its own copy of dynamically allocated memory.

By default, C++ provides a **default copy constructor**, which performs a **shallow copy**. However, if you want to customize the copying process or perform a deep copy, you need to define a **manual copy constructor**.

Your Code Explained

1. Class Definition

```
class Teacher {
public:
    // Attributes
    string name;
    string dept;
    string subject;
    double salary;

    // Parameterized constructor
    Teacher(string name, string dept, string subject, double salary) {
        this->name = name;
        this->dept = dept;
        this->subject = subject;
        this->salary = salary;
    }

    // Manual copy constructor
    Teacher(Teacher &orObj) {
        cout << "I'm manual copy constructor" << endl;
        this->name = orObj.name;
```

```
        this→dept = orObj.dept;
        this→subject = orObj.subject;
        this→salary = orObj.salary;
    }

    // Output function
    void display() {
        cout << "Name: " << name << endl;
        cout << "Department: " << dept << endl;
        cout << "Subject: " << subject << endl;
        cout << "Salary: " << salary << endl;
    }
};
```

2. Main Function

```
int main() {
    // Create an object t1 using the parameterized constructor
    Teacher t1("Shradha", "Computer Science", "C++", 25000);

    // Create an object t2 using the manual copy constructor
    Teacher t2(t1);

    // Display details of t1 and t2
    t1.display();
    cout << endl << "****" << endl;
    t2.display();

    return 0;
}
```

Step-by-Step Execution

1. **Object `t1` Creation:**

- The `Teacher` object `t1` is created using the **parameterized constructor**.
- The attributes of `t1` are initialized with the provided values:

```
t1.name = "Shradha";
t1.dept = "Computer Science";
t1.subject = "C++";
t1.salary = 25000;
```

2. **Object `t2` Creation:**
   - The `Teacher` object `t2` is created using the **manual copy constructor**.
   - The copy constructor is invoked, and the message `"I'm manual copy constructor"` is printed.
   - The attributes of `t2` are copied from `t1`:

```
t2.name = t1.name;
t2.dept = t1.dept;
t2.subject = t1.subject;
t2.salary = t1.salary;
```

3. **Display Details:**
   - The `display()` method is called for both `t1` and `t2`.
   - The output shows that `t2` is an exact copy of `t1`.

Output of the Program

```
I'm manual copy constructor
Name: Shradha
Department: Computer Science
Subject: C++
Salary: 25000

****
Name: Shradha
```

```
Department: Computer Science
Subject: C++
Salary: 25000
```

Key Concepts

1. Manual Copy Constructor

- The manual copy constructor is defined as:

```
Teacher(Teacher &orObj) {
    cout << "I'm manual copy constructor" << endl;
    this->name = orObj.name;
    this->dept = orObj.dept;
    this->subject = orObj.subject;
    this->salary = orObj.salary;
}
```

- It takes a reference to another `Teacher` object (`orObj`) and copies its attributes to the new object (`this`).

2. Why Use a Manual Copy Constructor?

- **Customization:** You can customize how the copying process works.
- **Deep Copy:** If the class contains dynamically allocated memory (e.g., pointers), a manual copy constructor ensures that the new object gets its own copy of the memory (deep copy).

3. Default Copy Constructor

- If you don't define a manual copy constructor, C++ provides a **default copy constructor** that performs a **shallow copy**.
- A shallow copy copies the values of the attributes but does not create new memory for dynamically allocated data.

When to Use a Manual Copy Constructor?

1. **Dynamic Memory Allocation:**
    - If your class contains pointers or dynamically allocated memory, you need a manual copy constructor to perform a **deep copy**.
2. **Custom Logic:**
    - If you want to add custom logic during the copying process (e.g., logging, validation), you need a manual copy constructor.

Example: Deep Copy with Manual Copy Constructor

Suppose the `Teacher` class has a dynamically allocated attribute:

```
class Teacher {
public:
    string name;
    string dept;
    string subject;
    double salary;
    int *grades; // Dynamically allocated array

    // Parameterized constructor
    Teacher(string name, string dept, string subject, double salary, int *grades)
{
        this->name = name;
        this->dept = dept;
        this->subject = subject;
        this->salary = salary;
        this->grades = new int[5];
        for (int i = 0; i < 5; i++) {
            this->grades[i] = grades[i];
        }
    }

    // Manual copy constructor (deep copy)
    Teacher(Teacher &orObj) {
        cout << "I'm manual copy constructor" << endl;
```

```
        this→name = orObj.name;
        this→dept = orObj.dept;
        this→subject = orObj.subject;
        this→salary = orObj.salary;
        this→grades = new int[5]; // Allocate new memory
        for (int i = 0; i < 5; i++) {
            this→grades[i] = orObj.grades[i]; // Copy values
        }
    }

    // Destructor
    ~Teacher() {
        delete[] grades; // Free dynamically allocated memory
    }

    // Output function
    void display() {
        cout << "Name: " << name << endl;
        cout << "Department: " << dept << endl;
        cout << "Subject: " << subject << endl;
        cout << "Salary: " << salary << endl;
        cout << "Grades: ";
        for (int i = 0; i < 5; i++) {
            cout << grades[i] << " ";
        }
        cout << endl;
    }
};
```

Conclusion

- A **manual copy constructor** allows you to customize how objects are copied.
- It is essential when your class contains dynamically allocated memory or requires custom logic during copying.

- In your code, the manual copy constructor ensures that `t2` is an exact copy of `t1`.

4. Move Constructor (C++11)

A constructor that moves resources from a temporary (rvalue) object to a new object. Used for performance optimization.

Example:

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

class Car {
public:
    string brand;
    Car(string b) : brand(b) {}
    Car(Car &&c) { // Move Constructor
        brand = move(c.brand);
        cout << "Move Constructor Called. Brand: " << brand << endl;
    }
};

int main() {
    Car car1("Mercedes");
    Car car2 = move(car1); // Move constructor is called
    return 0;
}
```

Output:

```
Move Constructor Called. Brand: Mercedes
```

Analogy: Constructor as an Object's "Birth Certificate"

Imagine a constructor as the process of **registering a newborn baby**:

- The **default constructor** is like a baby born without any name given initially.
- The **parameterized constructor** is like a baby whose name is decided at birth.
- The **copy constructor** is like making a duplicate birth certificate with the same details.
- The **move constructor** is like transferring ownership of a property instead of copying it.

Conclusion

- Constructors **automate object initialization**, making code cleaner and more reliable.
- Understanding different types of constructors helps in writing efficient C++ programs.
- **Use constructors wisely** to manage resources and avoid redundant code.

Concept Behind Multiple Constructors (Constructor Overloading)

- **Same constructor name but different parameters.**
- **The compiler determines which constructor to call** based on the number and type of arguments passed during object creation.
- **Provides flexibility** to initialize objects in different ways.

Example: Multiple Constructors in C++

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

class Teacher {
public:
```

```
    string name;
    string dept;
    string subject;
    double salary;

    // **Default Constructor**
    Teacher() {
        name = "Unknown";
        dept = "Not Assigned";
        subject = "None";
        salary = 0.0;
    }

    // **Parameterized Constructor**
    Teacher(string n, string d, string s, double sl) {
        name = n;
        dept = d;
        subject = s;
        salary = sl;
    }

    // **Constructor with Partial Parameters**
    Teacher(string n, string d) {
        name = n;
        dept = d;
        subject = "General";
        salary = 30000; // Default salary
    }

    void display() {
        cout << "Name: " << name << endl;
        cout << "Department: " << dept << endl;
        cout << "Subject: " << subject << endl;
        cout << "Salary: " << salary << endl;
    }
};
```

```
int main() {
    Teacher t1;  // Calls Default Constructor
    Teacher t2("Dharamveer", "Mathematics", "Maths", 45000); // Calls Parame
terized Constructor
    Teacher t3("Amit", "Physics"); // Calls Partial Parameter Constructor

    cout << "Teacher 1 Details:\\n";
    t1.display();

    cout << "\\nTeacher 2 Details:\\n";
    t2.display();

    cout << "\\nTeacher 3 Details:\\n";
    t3.display();

    return 0;
}
```

🛠️ Key Takeaways:

✔ **Multiple constructors allow flexible object initialization.**

✔ **The constructor is chosen based on arguments passed at object creation.**

✔ **Useful when default values or different initialization scenarios are needed.**

Would you like me to add this to your notes? 🚀

The `this` Pointer in C++

What is `this`?

In C++, `this` is a **special pointer** available inside **non-static** member functions. It holds the **memory address of the calling object.**

Key Points:

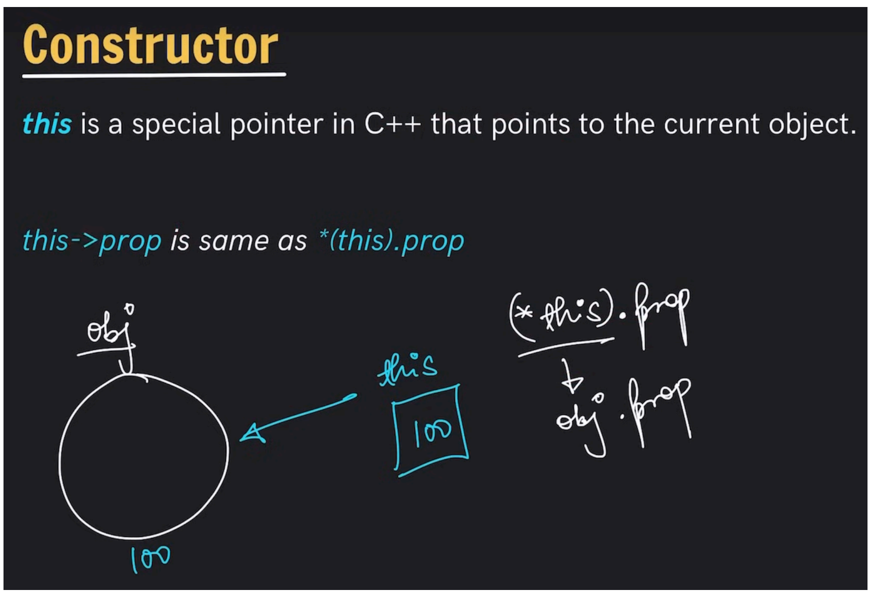
- `this` is an **implicit pointer** to the current object.

- It helps distinguish between **class members** and **function parameters**.
- It **cannot be modified** (a constant pointer).
- Used in **function chaining** by returning `this`.
- **Not available in static functions**, as they do not belong to any specific object.

Example: Using `this` to Access Members

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

class Teacher {
public:
    // Attributes
    string name;
    string dept;
    string subject;
    double salary;

    // Parameterized constructor
    Teacher(string name, string dept, string subject, double salary){
    // Left attributes belongs to object & right belongs to function parameter
    this->name = name;
    this->dept = dept;
    this->subject = subject;
    this->salary = salary;
    }

    //Output method
    void display(){
        cout <<"Name: " << name << endl;
        cout <<"Department: " << dept << endl;
        cout <<"Subject: " << subject << endl;
        cout <<"Salary: " << salary << endl;
```

```
    }

};

int main(){
    Teacher t1("Shradha", "Computer Science", "C++", 25000);

    t1.display();
    return 0;
}
```



Conclusion:

- `this` is useful for **object reference, function chaining, and resolving name conflicts**.
- Helps in maintaining **clarity and precision** in object-oriented programming.
- **Cannot be used in static member functions.**

Copy Constructor in C++

Core Concept:

A **copy constructor** is a special constructor in C++ used to create a **new object as a copy of an existing object**. It is called when:

- An object is initialized using another object (`Class obj2 = obj1;`).
- An object is passed **by value** to a function.
- An object is returned **by value** from a function.

If no copy constructor is explicitly defined, C++ provides a **default copy constructor**, which performs a **shallow copy**.

Example: Copy Constructor in Action

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

class Teacher {
public:
    string name;
    string dept;
    string subject;
    double salary;

    // Parameterized Constructor
    Teacher(string n, string d, string s, double sl) {
        name = n;
        dept = d;
        subject = s;
        salary = sl;
    }

    // Copy Constructor
    Teacher(const Teacher &t) {
        name = t.name;
        dept = t.dept;
        subject = t.subject;
```

```
        salary = t.salary;
        cout << "Copy Constructor Called!" << endl;
    }

    void getDetails() {
        cout << "Name : " << name << endl;
        cout << "Department : " << dept << endl;
        cout << "Subject : " << subject << endl;
        cout << "Salary : " << salary << endl;
    }
};

int main() {
    Teacher t1("Shardha Khapra", "Computer application", "C++", 30000);

    // Copy constructor is invoked
    Teacher t2(t1);
    t2.getDetails();
    return 0;
}
```

Key Takeaways:

✔️ **Copy constructor initializes an object using another object of the same class.**

✔️ **If no copy constructor is defined, C++ provides a default one.**

✔️ **Explicit copy constructors help in deep copying when dealing with dynamic memory.**

✔️ **Used in function argument passing and object returning.**

✔️ **Helps prevent unintended modifications when copying objects.**

If
you explicitly write your own copy constructor, the compiler **doesn't use the default one.** It uses **yours** instead.

🔷 Shallow Copy vs Deep Copy in C++ (Notes)

📌 What is Copy Constructor?

A **copy constructor** is a special constructor that creates a **new object as a copy of an existing object**. It is called when:

- A new object is initialized from an existing object (`Class obj2 = obj1;`).
- An object is passed **by value** to a function.
- An object is returned **by value** from a function.

```
ClassName(const ClassName &obj) {
    // Copy logic here
}
```

🔷 Shallow Copy

A **shallow copy** copies only the **memory addresses (pointers), not the actual data**.

This can cause **unexpected behavior** if the original object is modified or deleted.

🚩 Problem in Shallow Copy

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

class Teacher {
public:
    char* name;  // Pointer for dynamic memory allocation

    // Constructor
    Teacher(const char* n) {
```

```cpp
        name = new char[strlen(n) + 1];  // Allocating memory dynamically
        strcpy(name, n);
    }

    // ❌ Shallow Copy Constructor
    Teacher(const Teacher& t) {
        name = t.name;  // Only copies pointer, NOT data
    }

    void display() {
        cout << "Name: " << name << endl;
    }
};

int main() {
    Teacher t1("Shradha");
    Teacher t2 = t1; // Shallow Copy

    delete[] t1.name;  // ❌ Memory deleted for t1, affects t2

    t2.display();  // ❌ Undefined behavior (Dangling Pointer)

    return 0;
}
```

⚠️ What's Wrong?

- `t1.name` and `t2.name` point to **the same memory location**.
- Deleting `t1.name` makes `t2.name` a **dangling pointer**, leading to **undefined behavior**.

🔷 Deep Copy (Solution to Shallow Copy)

A **deep copy** creates a **new copy of the data** in a separate memory location.

✅ Corrected Example: Deep Copy Constructor

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

class Teacher {
public:
    char* name;  // Pointer for dynamic memory allocation

    // Constructor
    Teacher(const char* n) {
        name = new char[strlen(n) + 1];
        strcpy(name, n);
    }

    // ✅ Deep Copy Constructor
    Teacher(const Teacher& t) {
        name = new char[strlen(t.name) + 1];  // Allocating new memory
        strcpy(name, t.name);  // Copy actual data
    }

    void display() {
        cout << "Name: " << name << endl;
    }

    ~Teacher() {  // Destructor to free allocated memory
        delete[] name;
    }
};

int main() {
    Teacher t1("Shradha");
    Teacher t2 = t1; // Deep Copy

    delete[] t1.name;  // Only affects t1
```

```
    t2.display();  // ✅ t2 still has its own memory

    return 0;
}
```

✔️ Why Does Deep Copy Work?

- **Allocates new memory** and **copies actual data** instead of just the pointer.
- Prevents **dangling pointer issues**.
- `t2` has its own **separate copy of data**.

🔷 Why is Deep Copy Important in Dynamic Memory Allocation?

Dynamic memory allocation (`new`) requires **manual management** of memory.

If multiple objects **share the same memory location**, **deletion** of one object **invalidates** the others.

Using **deep copy ensures each object has its own memory**, preventing **crashes** and **unwanted modifications**.

🔷 Key Differences: Shallow Copy vs Deep Copy

Feature	Shallow Copy 🛑	Deep Copy ✅
Memory Allocation	Copies only **pointer**	Allocates **new memory**
Data Sharing	**Same** memory is shared	**Separate** memory for each object
Risk of Deletion	**Yes** (Dangling Pointer)	**No** (Each object is independent)
Use Case	Works for **non-pointer** variables	Required for **dynamic memory allocation**

🔷 Best Practices to Avoid Shallow Copy Issues

✅ Always use a **deep copy constructor** if your class **allocates dynamic memory**.

✅ **Override the assignment operator (`operator=`)** if your class has **pointers**.

✅ Use **smart pointers (`std::unique_ptr`, `std::shared_ptr`)** for automatic memory management.

🔷 Alternative: Smart Pointers (No Need for Deep Copy)

Instead of managing memory manually, **use smart pointers** like `std::unique_ptr`.

This **automatically manages memory** and prevents memory leaks.

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

class Teacher {
public:
    shared_ptr<string> name;  // Smart Pointer

    Teacher(string n) {
        name = make_shared<string>(n);
    }

    void display() {
        cout << "Name: " << *name << endl;
    }
};

int main() {
    Teacher t1("Shradha");
    Teacher t2 = t1; // ✅ Safe Copy (No memory issues)

    t1.display();
    t2.display();

    return 0;
}
```

✔️ **No need for copy constructor**

✔️ **No need for destructor**

✔️ **No dangling pointer issues**

🎯 Conclusion

- **Shallow Copy:** Copies only memory address (⚠️ Unsafe for dynamic memory).
- **Deep Copy:** Allocates **new memory** and **copies actual data** (✅ Safe).
- **Use deep copy** if your class has **dynamically allocated memory**.
- **Smart Pointers (`std::unique_ptr`, `std::shared_ptr`)** are a modern alternative.

🔷 Destructor in C++

📌 What is a Destructor?

A **destructor** is a special member function in C++ that **automatically gets called when an object goes out of scope or is deleted**. It is primarily used to **release resources** (like dynamically allocated memory, file handles, etc.) before an object is destroyed.

📌 Key Features of Destructors

✔️ **Same name as the class**, but prefixed with `~` (tilde).

✔️ **No parameters** and **no return type**.

✔️ **Called automatically** when the object is destroyed.

✔️ Used to **free memory, close files, or release other resources**.

✔️ If not defined, **C++ provides a default destructor** (but it does not free dynamically allocated memory).

🔷 Syntax of Destructor

```
class ClassName {
public:
```

```
    ~ClassName() {
        // Destructor code (cleanup)
    }
};
```

🔷 Example 1: Destructor in Action

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

class Example {
public:
    // Constructor
    Example() {
        cout << "Constructor is called!" << endl;
    }

    // Destructor
    ~Example() {
        cout << "Destructor is called!" << endl;
    }
};

int main() {
    Example obj;  // Object created → Constructor runs
    cout << "Inside main function" << endl;

    return 0;  // Object goes out of scope → Destructor runs automatically
}
```

✅ Output

```
Constructor is called!
Inside main function
```

```
Destructor is called!
```

🔷 Why Do We Need a Destructor?

- To **free dynamically allocated memory (`new` keyword)**
- To **close file handles or network connections**
- To **prevent memory leaks** in large applications

🔷 Example 2: Destructor Releasing Dynamically Allocated Memory

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

class Teacher {
public:
    string* name;

    // Constructor
    Teacher(string n) {
        name = new string(n);  // Allocating memory dynamically
        cout << "Constructor: Memory allocated for " << *name << endl;
    }

    // Destructor
    ~Teacher() {
        cout << "Destructor: Memory freed for " << *name << endl;
        delete name;  // Freeing memory
    }
};

int main() {
    Teacher t1("Shradha"); // Constructor runs
```

```
    return 0; // Destructor runs automatically
}
```

✅ Output

```
Constructor: Memory allocated for Shradha
Destructor: Memory freed for Shradha
```

🔷 Key Differences: Constructor vs Destructor

Feature	Constructor ✅	Destructor ❌
Purpose	Initializes an object	Cleans up an object
Name	Same as class name	Prefixed with `~`
Parameters	Can have parameters	**No parameters**
Return Type	No return type	No return type
Automatic Call	Called when an object is created	Called when an object is destroyed

🔷 Important Points About Destructors

✅ Only **one destructor per class** (no overloading).

✅ **Cannot be inherited**, but **base class destructor is called in derived class.**

✅ **Order of destruction is reverse** of construction (last created object gets destroyed first).

🔷 Example 3: Destructor in Inheritance

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

class Base {
public:
    Base() { cout << "Base Constructor" << endl; }
```

```
    ~Base() { cout << "Base Destructor" << endl; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor" << endl; }
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Derived obj;
    return 0;
}
```

✅ Output

```
Base Constructor
Derived Constructor
Derived Destructor
Base Destructor
```

✔️ **Base class constructor is called first**, but **its destructor is called last** (reverse order).

🎯 Conclusion

- A **destructor** is called automatically when an object **goes out of scope** or **is deleted**.
- **Used to free resources** like memory, files, and connections.
- **Cannot be overloaded** and has **no parameters**.
- **Destruction follows reverse order of construction** in inheritance.