Class – 10
C++ OOP class – 03

Constructors

A **constructor** is a special member function that is automatically invoked when an object of a class is created.

It initializes the object's data members so the object starts in a valid state.

The name of a constructor is always the same as the class name and it has no return type.

whenever you create an object like `Student s;`, the constructor with the same name as the class runs automatically.

You don't need to call it manually. It executes once for each object and sets the initial values of the attributes.

```
//constructor -> a special function that is automatically called when an obj
is created
//syntax
//classname() { ... }


Student() //initializes an obj
{
    cout<<"Default Constructor called"<<endl;


    name = "Othoy";
    age = 21;
    roll = 1087;
```

```
    cin>>name;
    cin>>age;
    cin>>roll;

    cout<<"name : "<<name<<endl;
    cout<<"age : "<<age<<endl;
    cout<<"roll : "<<roll<<endl;
}
```

This is a **custom default constructor**.

"Default" means it takes no parameters.

Here, the object `Student s;` triggers this constructor immediately.

It prints a message, sets default values for `name`, `age`, and `roll`, and even takes input from the user to modify them.

Inside it, initialization and input/output both happen.

So every time you create a Student object, this block runs and automatically prepares the object.

```
Student s;// built-in default constructor gets called when there's no other constructor
```

This line shows what happens if you don't define your own constructor.

C++ will still create an object using a compiler-generated default constructor.

When you define your own custom constructor, it replaces the compiler's version.

```
void setter(string n,int a,int r)
{
    name = n;
    age = a;
    roll = r;
}
```

This setter manually assigns values to attributes after an object already exists.

Unlike a constructor, which runs automatically, you must call `setter()` yourself.

It shows how ordinary functions can still modify objects, but they lack the automatic behavior that constructors provide.

```
Student(string n="",int a=0,int r=0)
{
    cout<<"Parameterized Constructor called"<<endl;

    name = n;
    age = a;
    roll = r;
}
```

This is a **parameterized constructor with default arguments**.

It accepts parameters and uses them to initialize the object.

But because each argument has a default value, it can also be used with no parameters—acting like a default constructor too.

For example, `Student s("olivia",23,7888);` passes real values, while `Student ss;` uses all defaults.

```
Student(string n,int a,int r) : roll(122),age(a),name(n)
{
    cout<<"Parameterized Constructor called"<<endl;
    cin>>name;
}
```

This uses an **initialization list**.

The section after the colon `:` initializes the data members before the constructor body runs.

For instance, `roll(122)` sets the roll number right at the moment of object creation, not later.

```
Student()
{
    cout<<"Default Constructor called"<<endl;
    name = "Othoy";
    age = 21;
    roll = 1087;
}

Student(string n,int a,int r)
{
    cout<<"Parameterized Constructor called"<<endl;
    name = n;
```

```
    age = a;
    roll = r;
}
```

This demonstrates **constructor overloading**.

Two constructors have the same name (`Student`) but different parameter lists.

The compiler automatically decides which one to call depending on the way the object is created.

If you write `Student s;`, the default constructor runs.

If you write `Student s("olivia",23,7888);`, the parameterized one runs.

```
void f(int x) //copy -> doesn't change the original variable
{
    cout<<x<<endl;
    x = 12;
    cout<<x<<endl;
}

void f(int &x) // reference parameter
{
    cout<<x<<endl;
    x = 12;
    cout<<x<<endl;
}
```

Here two functions `f()` show the difference between pass by value and pass by reference.

In the first one, `x` is a copy, so changing it doesn't affect the original variable in main.

In the second one, `x` is a reference, meaning it refers to the original variable, so any change updates the real one too.

References are like alternate names for the same memory location.

```
void fun(Student obj) //copy object created here
{
    obj.display();
}
```

When an object is passed to a function by value, a copy of that object is made.

The copy process happens through the copy constructor.

Even though the copy constructor is not explicitly defined in your code, the compiler provides one automatically.

It duplicates all the data members of the original object into the new temporary one.

That's why when `fun(s);` is called, the function receives a completely separate object `obj`.

If you modify `obj`, it won't affect the original `s`.