Constructors in C++

- A constructor is a **special member function** of a class that is automatically called when an object of the class is created.
- Its name is the **same as the class name**.
- It has **no return type** (not even `void`).

Default Constructor

- A constructor **without parameters**.
- It initializes objects with **default values**.
- If you don't define any constructor, the compiler provides a default constructor automatically.

Syntax:

```
class Student {
    int age;
    string name;

public:
    // Default constructor
    Student() {
        age = 0;
        name = "Unknown";
    }

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

int main() {
    Student s1; // Default constructor is called
    s1.display();
    return 0;
}
```

Output:

```
Name: Unknown, Age: 0
```

Parameterized Constructor

- A constructor that takes **arguments/parameters** to initialize an object with user-defined values.

Syntax:

```
class Student {
    int age;
    string name;

public:
```

```
    // Parameterized constructor
    Student(string n, int a) {
        name = n;
        age = a;
    }

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

int main() {
    Student s1("Neeraj", 23);  // Parameterized constructor is called
    Student s2("Amit", 21);

    s1.display();
    s2.display();
    return 0;
}
```

Output:

```
Name: Neeraj, Age: 23
Name: Amit, Age: 21
```

Key Points

- Constructors are **automatically invoked** when an object is created.
- You can have **multiple constructors** in a class (Constructor Overloading).
- If you define a parameterized constructor but still want a default one, you must **explicitly define it**.

Constructor Overloading

- When a class has **multiple constructors** with **different parameter lists**.
- Helps create objects in **different ways**.
- This is an example of **compile-time polymorphism**.

Example:

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

class Student {
    string name;
    int age;

public:
    // Default constructor
    Student() {
        name = "Unknown";
        age = 0;
    }

    // Parameterized constructor (1 parameter)
    Student(string n) {
```

```
        name = n;
        age = 0;
    }

    // Parameterized constructor (2 parameters)
    Student(string n, int a) {
        name = n;
        age = a;
    }

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

int main() {
    Student s1;                // Calls default constructor
    Student s2("Neeraj");      // Calls constructor with 1 parameter
    Student s3("Amit", 21);    // Calls constructor with 2 parameters

    s1.display();
    s2.display();
    s3.display();
    return 0;
}
```

Output:

```
Name: Unknown, Age: 0
Name: Neeraj, Age: 0
Name: Amit, Age: 21
```

Constructor with Default Arguments

- A constructor that has **default values for its parameters**.
- This avoids writing multiple constructors, because **one constructor can handle multiple cases**.

Example:

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

class Student {
    string name;
    int age;

public:
    // Constructor with default arguments
    Student(string n = "Unknown", int a = 0) {
        name = n;
        age = a;
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
```

```
int main() {
    Student s1;                 // Uses default arguments -> "Unknown", 0
    Student s2("Neeraj");       // Age defaults to 0
    Student s3("Amit", 21);     // Both values provided

    s1.display();
    s2.display();
    s3.display();
    return 0;
}
```

Output:

```
Name: Unknown, Age: 0
Name: Neeraj, Age: 0
Name: Amit, Age: 21
```

Key Differences

Feature	Constructor Overloading	Constructor with Default Arguments
Definition	Multiple constructors with different parameter lists	Single constructor with parameters having default values
Code Length	Longer (multiple constructors needed)	Shorter (one constructor handles all cases)
Flexibility	More flexible, can define custom initialization for each case	Less flexible but concise

Implicit Call of Constructor

- When you create an object **without directly calling the constructor**, the constructor is called **implicitly (automatically)**.
- This is the **normal way** we create objects.

Example:

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

class Student {
public:
    Student() {
        cout << "Default constructor called" << endl;
    }
};

int main() {
    Student s1;   //  Implicit call
    return 0;
}
```

Output:

```
Default constructor called
```

Explicit Call of Constructor

- A constructor can also be called **explicitly** using the class name.
- Useful when you want to create a **temporary object** or reinitialize an object.

Example:

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

class Student {
public:
    Student() {
        cout << "Default constructor called" << endl;
    }
};

int main() {
    Student s1 = Student();   // 👈 Explicit call
    Student s2;               // 👈 Implicit call

    return 0;
}
```

Output:

```
Default constructor called
Default constructor called
```

Both work, but the **style of calling is different**.

Dynamic Initialization of Objects

- **Dynamic Initialization** means initializing objects at **runtime using variables or user input**.
- Constructors can take values from **expressions, variables, or user input**.

Example:

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

class Student {
    string name;
    int age;
public:
    // Parameterized constructor
    Student(string n, int a) {
        name = n;
        age = a;
    }

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};
```

```
int main() {
    string n;
    int a;

    cout << "Enter name: ";
    cin >> n;
    cout << "Enter age: ";
    cin >> a;

    // Dynamic initialization with user input
    Student s1(n, a);
    s1.display();

    return 0;
}
```

Input / Output Example:

```
Enter name: Neeraj
Enter age: 23
Name: Neeraj, Age: 23
```

Summary Table

Concept	Meaning	Example
Implicit Call	Constructor automatically called when object is created	`Student s1;`
Explicit Call	Constructor called directly using class name	`Student s1 = Student();`
Dynamic Initialization	Object initialized with values at runtime	`Student s1(n, a);`

Copy Constructor

- A **copy constructor** is a special constructor which is used to create a **new object as a copy of an existing object**.
- Syntax:
- `ClassName(const ClassName &obj) { ... }`

When is Copy Constructor Called?

- ✓ When an object is initialized from another object of the same class.
- ✓ `Student s2(s1);  // Copy constructor called`
- ✓ When an object is passed **by value** to a function.
- ✓ When a function returns an object **by value**.

Example:

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

class Student {
    string name;
```

```
    int age;

public:
    // Parameterized constructor
    Student(string n, int a) {
        name = n;
        age = a;
    }

    // Copy constructor
    Student(const Student &s) {
        cout << "Copy constructor called!" << endl;
        name = s.name;
        age = s.age;
    }

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

int main() {
    Student s1("Neeraj", 23);  // Normal constructor
    Student s2(s1);            // Copy constructor (s2 is a copy of s1)

    s1.display();
    s2.display();
    return 0;
}
```

Output:

```
Copy constructor called!
Name: Neeraj, Age: 23
Name: Neeraj, Age: 23
```

Destructor

- A **destructor** is a special member function that is automatically called when an object goes out of scope or is deleted.
- Its purpose is to **free resources** (like memory, file handles, connections).
- It has the **same name as the class but with a ~ (tilde) sign** in front.
- No parameters, no return type, and cannot be overloaded.

Syntax:

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

Example:

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

class Student {
    string name;
```

```
public:
    Student(string n) {
        name = n;
        cout << "Constructor called for " << name << endl;
    }

    ~Student() {
        cout << "Destructor called for " << name << endl;
    }
};

int main() {
    Student s1("Neeraj");
    Student s2("Amit");

    cout << "End of main function." << endl;
    return 0;
}
```

Output:

```
Constructor called for Neeraj
Constructor called for Amit
End of main function.
Destructor called for Amit
Destructor called for Neeraj
```

Notice: Destructors are called in **reverse order of construction**.

Quick Comparison

Feature	Copy Constructor	Destructor
Purpose	Creates a new object as a copy of another object	Destroys object and frees resources
Syntax	`ClassName(const ClassName &obj)`	`~ClassName()`
Called When	New object created from existing object	Object goes out of scope / deleted
Overloading	Can be overloaded	Cannot be overloaded