1. Introduction

In C++, constructors and destructors are special member functions of a class that handle **object creation** and **object destruction**, respectively. They provide **automatic initialization and cleanup**, making programs easier to write and manage.

- **Constructor:** Automatically called when an object is created.
- **Destructor:** Automatically called when an object is destroyed.

---

2. Importance of Constructors and Destructors

- Automatically initialize data members
- Prevent repetitive code
- Ensure proper cleanup of resources
- Reduce programming errors
- Useful for dynamic memory allocation

---

3. Constructor

A constructor is a special member function with the **same name as the class** and **no return type**, not even `void`.

Key Points:

- Called automatically during object creation
- Can be overloaded
- Cannot return a value
- Can have default or parameterized forms

---

4. Syntax of Constructor

```
class ClassName {
  public:
    ClassName() { // body of constructor }
};
```

Example

```
class Student {
  public:
    int roll;
    Student() { // constructor
      roll = 0;
    }
};
```

Here, when an object of Student is created, roll is automatically initialized to 0.

---

5. Types of Constructors

5.1 Default Constructor

- Constructor with no parameters
- Used to provide default values

```
class Student {
  public:
    int roll;
    Student() { roll = 0; }
};
```

5.2 Parameterized Constructor

- Constructor with parameters
- Used to initialize objects with specific values

```
class Student {
  public:
    int roll;
    Student(int r) { roll = r; }
};
```

5.3 Copy Constructor

- Used to create a new object as a copy of an existing object
- Syntax: ClassName(const ClassName &obj);

```
Student(const Student &s) {
  roll = s.roll;
}
```

---

6. Constructor Overloading

- Multiple constructors with different parameter lists
- Compiler chooses the correct constructor based on arguments

```
class Student {
  public:
    Student() { }
    Student(int r) { }
    Student(int r, string n) { }
};
```

---

7. Dynamic Constructors

Constructors can also be used with **dynamic memory allocation** to initialize pointers and arrays at runtime.

```
class Array {
   int *arr;
   int size;
   public:
      Array(int s) {
         size = s;
         arr = new int[size];
      }
};
```

---

8. Destructor

A destructor is a special member function that is called **automatically when an object goes out of scope or is deleted**.

Key Points:

- Has the **same name as the class** prefixed with ~
- No return type and no parameters
- Used to release memory or resources
- Only **one destructor per class** (cannot be overloaded)

---

9. Syntax of Destructor

```
class ClassName {
   public:
      ~ClassName() { // body of destructor }
};
```

Example

```
class Student {
   public:
      ~Student() { cout << "Object destroyed"; }
};
```

When a Student object is destroyed, the destructor prints the message.

---

10. Use of Destructors

- Free dynamically allocated memory
- Close files
- Release system resources
- Perform cleanup operations

```
class File {
   FILE *fp;
   public:
      File() { fp = fopen("data.txt","w"); }
      ~File() { fclose(fp); }
};
```

---

11. Constructor vs Destructor

Feature	Constructor	Destructor
Called	When object is created	When object is destroyed
Name	Same as class	~ + class name
Return type	None	None
Parameters	Can have	Cannot have
Overloading	Yes	No

---

12. Order of Execution

1. Base class constructors are called first
2. Derived class constructors are called after base class
3. Destructors are called in **reverse order**
    - Derived class destructor first
    - Base class destructor last

---

13. Constructor and Destructor in Inheritance

Example

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

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

Output:

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

14. Dynamic Memory and Destructors

- Dynamically allocated memory using `new` must be released using `delete` in destructors
- Prevents **memory leaks**

```
class Array {
   int *arr;
   public:
      Array(int size) { arr = new int[size]; }
      ~Array() { delete[] arr; }
};
```

15. Advantages of Constructors

- Automatic initialization
- Avoid repetitive code
- Supports multiple initialization through overloading
- Enhances readability and maintainability

16. Advantages of Destructors

- Automatic cleanup
- Prevents memory leaks
- Ensures resources are released
- Simplifies program design

17. Common Mistakes

- Forgetting to use destructor for dynamic memory
- Overloading destructors (not allowed)
- Not using constructors for initialization
- Calling destructors explicitly (avoid unless necessary)

18. Best Practices

- Always initialize objects using constructors
- Use destructors for releasing memory and resources
- Prefer default and parameterized constructors for flexibility
- Avoid unnecessary use of destructors for stack-allocated objects

19. Applications

- Resource management (files, memory)
- Automatic initialization of objects
- Object cleanup in classes with dynamic memory
- Safe handling of inheritance hierarchies

---

20. Conclusion

Constructors and destructors are **fundamental for object management in C++**. Constructors ensure that objects are properly initialized, while destructors ensure resources are released when objects are no longer needed. Proper understanding and use of constructors and destructors improve **program reliability, memory management, and maintainability** in C++ programming.