Destructor in C++?

🔷 What is a Destructor in C++?

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

It is used to:

- **Clean up resources** (memory, file handles, network connections, etc.)
- Prevent **memory leaks**
- it is an instance member function that is invoked automatically whenever object is going to be destroyed.
- it is the last function that is going to be called before an object is destroyed.
- ye hamesha ek hi hota hai
- koi return type nhi hota
- ye bhi by default create rehta in class me
- ye by default jo jo chize initialized hai unko delete kr deta jo jo dynamically diye hote

🔧 Syntax

```
cpp
CopyEdit
~ClassName();



defaultly aisi dikhte
~class(){
delete balance ;
}
```

use case :

ye storage ko release kr deta jb use nhi hota to like jaise printer use kr rhe ab no lopgner use hai to release kr do taki dusre use kr ske

program stak me run hota to jo jo stack me hota int a , int b , strin name to y to bad me execution ke sath release ho jata

but jisko dynamicly memeory diye hai like e *p = x ;   *ptr= student   ; to ye sb to heap me hote to inko bhi release krwna hota

ye variabel ko delete nhi krta just unke location ko release kr ste

- It has the **same name as the class**, preceded by a **tilde (~)**.
- It **does not take any arguments**.
- It **does not return anything** (not even `void`).
- There is **only one destructor per class** (no overloading).

Sure! Let's explore **destructors** in C++ in **complete detail** — including definition, syntax, examples, real-life analogy, and interview-focused points.

🔷 What is a Destructor in C++?

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

It is used to:

- **Clean up resources** (memory, file handles, network connections, etc.)
- Prevent **memory leaks**

🔧 Syntax

```
~ClassName();
```

- It has the **same name as the class**, preceded by a **tilde (~)**.

- It **does not take any arguments**.
- It **does not return anything** (not even `void`).
- There is **only one destructor per class** (no overloading).

🔷 Real-Life Analogy

Imagine a **classroom** (object).

When the class ends (object is destroyed), the **cleaner (destructor)** comes in and:

- Switches off the lights
- Closes windows
- Wipes the board

Similarly, in programming, when an object is no longer needed, the destructor ensures everything is **cleaned up** properly.

🔷 Example

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

class Demo {
public:
    Demo() {
        cout << "Constructor called!" << endl;
    }

    ~Demo() {
        cout << "Destructor called!" << endl;
    }
};

int main() {
    Demo d1;  // Constructor is called here
    // Destructor will be called automatically when d1 goes out of scope
}
```

Output:

```
Constructor called!
Destructor called!
```

🔷 Important Use Case – Dynamic Memory

```
class Memory {
    int* ptr;

public:
    Memory() {
        ptr = new int[5];  // Dynamic memory
        cout << "Memory allocated!" << endl;
    }

    ~Memory() {
        delete[] ptr;  // Free the memory
        cout << "Memory deallocated!" << endl;
    }
};

int main() {
    Memory m1;
}
```

Output:

```
Memory allocated!
Memory deallocated!
```

Without a destructor, this would cause a **memory leak**!

🔷 When is Destructor Called?

- When object **goes out of scope**
- When object is **explicitly deleted** (`delete` keyword for dynamic object)
- At the **end of the program** for static/global objects

🔷 Can We Call Destructor Manually?

Yes, but it's **not recommended** unless you have a very good reason:

```
Demo d;
d.~Demo();  // Manual call (bad practice usually)
```

🔷 Constructor vs Destructor

Feature	Constructor	Destructor
Name	Same as class	Same as class with `~`
Purpose	Initializes object	Cleans up object
Parameters	Can have parameters	No parameters
Return type	No return type	No return type
Overloading	Can be overloaded	Cannot be overloaded
Called	When object is created	When object is destroyed

🔷 Interview Points to Remember

1. **Destructor is automatically called** when object goes out of scope.
2. Only **one destructor per class** — no overloading.
3. Use destructors to **release dynamically allocated memory** or **close resources**.

4. Destructors are **crucial in preventing memory leaks** in large applications.
5. If a class has a **pointer**, always define a destructor to `delete` that memory.
6. If a class has a **constructor but no destructor**, the compiler provides a **default destructor** — but it won't delete dynamically allocated memory!

🔷 Example with `new` and `delete`

```
class Test {
public:
    Test() {
        cout << "Object Created\n";
    }

    ~Test() {
        cout << "Object Destroyed\n";
    }
};

int main() {
    Test* t = new Test();  // Constructor called
    delete t;              // Destructor called manually via delete
}
```

🔷 Summary

- Destructor is **automatic cleanup code**.
- Helps manage **resources safely**.
- Critical in **C++ for memory management**, especially with `new` / `delete`.

Would you like to see a **real-world project scenario** using destructor (like in file handling or class with database connection)?

📌 Interview Insights

- **Q:** Can an empty class have size 0 in C++?

  A: No, it will be **at least 1 byte** to give it a distinct address.

- **Q:** Why does the compiler add 1 byte to an empty class?

  A: So each object has a **unique memory address**.

- **Q:** What increases the size of an empty class?

  A: Adding **virtual functions**, **multiple inheritance**, or **non-static members**.

🧠 Interview Tips:

1. **Constructor vs Method** – Constructor auto-calls; method you call manually.
2. Can't have `return` in constructor.
3. **Can't be `virtual`**, but **destructors can**.
4. Can have **default arguments**.
5. **Copy constructor must take const reference**.

- **Can a constructor be private?**
  - Yes, used in **Singleton Design Pattern**.
- **Constructor vs Method:**

Constructor	Method (Function)
Called automatically	Called manually
No return type	Has return type
Same name as class	Any name

- If **no constructor** is defined, compiler provides a **default constructor**.
- **Constructor Overloading** allows creating multiple constructors with different parameters.

Great question — you're noticing something very important about how **object lifetimes** work in C++.

🔁 Why is Destructor Called in Reverse Order?

In your program:

```
customer c1("1");
customer c2("2");
customer c3("3");
```

- You created three objects: `c1`, `c2`, and `c3` — **in that order**.
- Their **constructors** were called **in the order of creation** (1 → 2 → 3).
- But their **destructors** are called **in reverse** (3 → 2 → 1).

🔍 Why Does This Happen?

👉 Stack-Based Storage

Objects like `c1`, `c2`, and `c3` are **local variables**, so they are created on the **stack**. The stack is a **Last In, First Out (LIFO)** structure:

- **Last created object is destroyed first.**

It's just like a **stack of plates** — you put plate 1, then 2, then 3 on top. When cleaning up, you take plate 3 first, then 2, then 1.

🧠 Why is this reverse destruction important?

✅ To Handle Dependencies Safely

Imagine this case:

- `Object A` uses `Object B` inside it.

- If `Object B` is destroyed **before** `A`, you may get a **crash** (because `A` is accessing something that's already destroyed).

So C++ makes sure:

> Destroy the most recently created objects first — so that any older objects still remain alive if needed.

🔧 Analogy

Think of a **hotel check-in desk**:

- First customer to check-in is `c1`.
- Then `c2`.
- Then `c3`.

Now during checkout, `c3` checks out first (most recent).

Why? Because otherwise, the person who checked in last might get blocked by earlier ones.

✅ Output of Your Code:

```
constructor is call 1
constructor is call 2
constructor is call 3
destructor is called   3
destructor is called   2
destructor is called   1
```

🧪 Pro Tip:

If you want to **create objects dynamically using** `new`, then you must **manually delete** them in reverse too.

Would you like me to also explain how destruction works with `new` (heap-allocated objects) and in class composition (has-a relationships)?