

# Destructors in C++

Last Updated : 01 May, 2025



Destructor is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

## Syntax

Destructors are automatically present in every C++ class but we can also redefine them using the following syntax.

```
~className(){  
    // Body of destructor  
}
```

where, tilda(~) is used to create destructor of a **className**.

Just like any other member function of the class, we can define the destructor outside the class too:

```
className {  
    public:  
        ~className();  
}  
  
class-name :: ~className() {  
    // Desctructor Body  
}
```

But we still need to at least declare the destructor inside the class.

## Examples of Destructor

The below programs demonstrate the behaviour of the destructor in different cases:

### Example 1

The below code demonstrates the automatic execution of constructors and destructors when objects are created and destroyed, respectively.

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

```

3
4  class Test {
5  public:
6
7      // User-Defined Constructor
8      Test() {
9          cout << "Constructor Called"
10             << endl;
11     }
12
13     // User-Defined Destructor
14     ~Test() {
15         cout << "Destructor Called"
16            << endl;
17     }
18 };
19 main() {
20     Test t;
21
22     return 0;
23 }

```

## Output

```

Constructor Called
Destructor Called

```

## Example 2

The below C++ program demonstrates the number of times constructors and destructors are called.

```

1  ↔
6  int Count = 0;
7  class Test {
8  public:
9      Test(){
10
11         // Number of times constructor is called
12         Count++;

```

```

13         cout << "No. of Object created: "
14             << Count << endl;
15     }
16     ~Test() {
17
18         // It will print count in decending order
19         cout << "No. of Object destroyed: " << Count
20             << endl;
21         Count--;
22     }
23 };
24
25 int main() {
26     Test t, t1, t2, t3;
27     ↔

```

## Output

```

No. of Object created: 1
No. of Object created: 2
No. of Object created: 3
No. of Object created: 4
No. of Object destroyed: 4
No. of Object destroyed: 3
No. of Object destroyed: 2
No. of Object destroyed: 1

```

**Note:** Objects are destroyed in the reverse order of their creation. In this case, *t3* is the first to be destroyed, while *t* is the last.

## When do we need to write a user-defined destructor?

If we do not write our own destructor in class, the compiler creates a default destructor for us. **The default destructor works fine unless we have dynamically allocated memory or pointer in class.** When a class contains a pointer to memory allocated in the class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leaks.

### Example:

```

1  <=>
6  class MyClass {
7  private:
8
9      // Pointer to dynamically
10     // allocated memory
11     int* data;
12
13 public:
14     MyClass(int value) {
15         data = new int;
16         *data = value;
17         cout << *data << endl;
18     }
19
20     // User-defined destructor: Free
21     // the dynamically allocated memory
22     ~MyClass() {
23
24         // Deallocate the dynamically
25         // allocated memory
26         delete data;
27         cout << "Destructor: Memory deallocated";
28     }
29 };
30
31 int main() {
32     MyClass obj1(10);
33 <=>

```

## Output

```

10
Destructor: Memory deallocated

```

In the above example, when the object is destroyed, the destructor releases the dynamically allocated resources, which in this case is the pointer.

## Characteristics of a Destructor

All the points mentioned below show the characteristics of a destructor:

- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence, destructor cannot be overloaded.
- It cannot be declared static or const.
- Destructor neither requires any argument nor returns any value.
- It is automatically called when an object goes out of scope.
- Destructor release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of an object creation.

## When is the destructor called?

A destructor function is called automatically when the object goes out of scope or is deleted. Following are the cases where destructor is called:

1. Destructor is called when the function ends.
2. Destructor is called when the program ends.
3. Destructor is called when a block containing local variables ends.
4. Destructor is called when a delete operator is called.

## How to call destructors explicitly?

**Destructor** can also be called explicitly for an object. We can call the destructors explicitly using the following statement:

```
object_name.~class_name()
```



### Constructors and Destructors | Video

[Visit Course ↗](#)[Comment](#)[More info ▼](#)[Advertise with us](#)[Next Article >](#)[Private Destructor in C++](#)

## Similar Reads

### C++ Programming Language

C++ is a computer programming language developed by Bjarne Stroustrup as an extension of the C language. It is known for its fast speed, low level memory management and is often taught as first programming language. It...

🕒 5 min read

---

#### C++ Overview



---

#### C++ Basics



---

#### C++ Variables and Constants



---

#### C++ Data Types and Literals



---

#### C++ Operators



---

#### C++ Input/Output



---

#### C++ Control Statements



---

#### C++ Functions



---

#### C++ Pointers and References

