Destructor Execution Order OOP

In Object-Oriented Programming (OOP), destructors are executed in reverse order of the constructors because of the **LIFO (Last In, First Out)** principle, which ensures that resources are properly released or cleaned up.

Key Reasons for Destructors Being Executed in Reverse Order:

1. **Resource Cleanup in Reverse Dependency Order**:
   - Objects are often constructed in a specific order, where one object might depend on another. When objects are destroyed, they need to be destroyed in the opposite order to ensure that an object dependent on another one is destroyed first. This prevents access to already destroyed objects, which could lead to undefined behavior.
   - For example:
     - If object **A** depends on object **B**, **A** must be destroyed before **B** to ensure that **B** is still valid when **A**'s destructor is called.

2. **Stack-Like Behavior (LIFO)**:
   - The construction and destruction process works like a stack. When objects are created, they are pushed onto a stack, and when they are destroyed, they are popped off the stack in reverse order.
   - This is a common behavior when allocating and deallocating resources, like memory, file handles, etc.

3. **Consistency with Constructor Execution**:
   - When you create an object with multiple sub-objects or base classes, the constructors for those sub-objects or base classes are called first, building the object from the ground up. When destroying the object, you reverse this process to properly dismantle it.
   - This ensures that the destructor of the derived class is called before the base class destructor.

Example to Illustrate Reverse Destructor Execution:

cpp

```cpp
#include <iostream>

class A {
public:
    A() { std::cout << "Constructor of A\n"; }
    ~A() { std::cout << "Destructor of A\n"; }
};

class B {
public:
    B() { std::cout << "Constructor of B\n"; }
    ~B() { std::cout << "Destructor of B\n"; }
};
```

```cpp
class C : public A {
public:
    B b;
    C() { std::cout << "Constructor of C\n"; }
    ~C() { std::cout << "Destructor of C\n"; }
};

int main() {
    C c;
    return 0;
}
```

Output:

```css
Constructor of A
Constructor of B
Constructor of C
Destructor of C
Destructor of B
Destructor of A
```

Explanation:

- **Construction Order**:
    1. First, the base class **A** is constructed (`` `Constructor of A` ``).
    2. Then, member object **B** is constructed (`` `Constructor of B` ``).
    3. Finally, the derived class **C** is constructed (`` `Constructor of C` ``).

- **Destruction Order (Reverse)**:
    1. The derived class **C** is destroyed first (`` `Destructor of C` ``).
    2. Then, member object **B** is destroyed (`` `Destructor of B` ``).
    3. Finally, the base class **A** is destroyed (`` `Destructor of A` ``).

By executing destructors in reverse order, we ensure the proper cleanup of resources and avoid potential access to already destroyed objects or resources.