Virtual Base Class in C++

A **virtual base class** is used in **multiple inheritance** to **avoid duplication of base class members** when the same base class is inherited more than once through different paths.

It ensures that **only one copy of the base class** is inherited, even if it appears multiple times in the inheritance hierarchy.

Problem Without Virtual Base Class:

Let's understand using an example:

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

class A {
public:
    int roll;
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main() {
    D obj;
    // obj.roll = 10; // Error: Ambiguity, which A to use (B::A or C::A)?
}
```

Here,

- `D` inherits from both `B` and `C`.
- Both `B` and `C` have their own copy of `A`.
- Hence, **two copies of `A`** exist in `D`, causing **ambiguity**.

Solution: Using Virtual Base Class

We can declare `A` as a **virtual base class** to remove duplication:

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

class A {
public:
    int roll;
};

class B : virtual public A {
};
```

```
class C : virtual public A {
};

class D : public B, public C {
};

int main() {
    D obj;
    obj.roll = 10;   // No ambiguity now
    cout << "Roll = " << obj.roll;
    return 0;
}
```

Output:

```
Roll = 10
```

Key Points:

- ✓ **Used in multiple inheritance** to avoid duplication of base class members.
- ✓ The **base class is shared** among all derived classes.
- ✓ Virtual base class ensures **single instance** of base class.
- ✓ It **removes ambiguity** in “diamond problem.”
- ✓ **Constructors** of virtual base classes are called **only once** — by the most derived class.

Constructor in Derived Class (C++)

When a class is derived from another class, **the base class constructor is called first**, followed by the **derived class constructor**.

This ensures that the base class part of the derived object is properly initialized **before** the derived class adds its own members.

Order of Constructor Execution:

1. **Base class constructor** executes first.
2. **Derived class constructor** executes next.

Basic Example:

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

class Base {
public:
    Base() {
        cout << "Base constructor called\n";
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived constructor called\n";
    }
};

int main() {
    Derived obj;
    return 0;
}
```

Output:

```
Base constructor called
Derived constructor called
```

Constructor with Parameters (Parameterized Constructor):

If the base class has a parameterized constructor,
the derived class must **explicitly call it** using an **initialization list**.

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

class Base {
    int a;
public:
    Base(int x) {
        a = x;
        cout << "Base constructor called with a = " << a << endl;
    }
};

class Derived : public Base {
    int b;
public:
    Derived(int x, int y) : Base(x) {   // Base constructor called
explicitly
        b = y;
        cout << "Derived constructor called with b = " << b << endl;
    }
};

int main() {
    Derived obj(10, 20);
    return 0;
}
```

Output:

```
Base constructor called with a = 10
Derived constructor called with b = 20
```

Constructor Call in Multiple Inheritance:

If a derived class inherits from multiple base classes,
then **base class constructors are called in the order of inheritance declaration** (from left to right).

```
class A {
public:
    A() { cout << "A constructor\n"; }
};

class B {
public:
    B() { cout << "B constructor\n"; }
};

class C : public A, public B {
public:
    C() { cout << "C constructor\n"; }
};

int main() {
    C obj;
}
```

Output:

```
A constructor
B constructor
C constructor
```

Constructor Call in Virtual Base Class:

When **virtual inheritance** is used,
the **virtual base class constructor is called only once**,
and it is called **by the most derived class**.

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

class A {
public:
    A() { cout << "A constructor\n"; }
};

class B : virtual public A {
public:
    B() { cout << "B constructor\n"; }
};

class C : virtual public A {
public:
    C() { cout << "C constructor\n"; }
};

class D : public B, public C {
public:
    D() { cout << "D constructor\n"; }
};

int main() {
    D obj;
    return 0;
}
```

Output:

```
A constructor
B constructor
C constructor
D constructor
```

Even though both `B` and `C` inherit `A`,
`A`'s constructor runs **only once** — due to **virtual base class**.

Initialization List in C++

An **Initialization List** is a special feature in C++ used to **initialize data members** of a class **before the constructor body executes**.

It comes **after the constructor's parameter list** and **before the body** — separated by a colon (`:`).

Example 1: Basic Use

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

class Student {
    int roll;
    string name;
public:
    // Initialization list used here
    Student(int r, string n) : roll(r), name(n) {
        cout << "Constructor called\n";
    }

    void show() {
        cout << "Roll: " << roll << ", Name: " << name << endl;
    }
};

int main() {
    Student s1(101, "Neeraj");
    s1.show();
    return 0;
}
```

Output:

```
Constructor called
Roll: 101, Name: Neeraj
```

Why Use Initialization List?

Initialization lists are preferred because:

1. **Efficiency** – avoids extra assignments inside constructor body.
2. **Required for `const` data members** (they must be initialized at declaration).

3. **Required for reference members (`&`)**.
4. **Used to call Base Class Constructors** in derived classes.

Example 2: Const and Reference Members

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

class Demo {
    const int a;
    int &b;
public:
    Demo(int x, int &y) : a(x), b(y) {   // must use initialization list
        cout << "Values initialized\n";
    }

    void display() {
        cout << "a = " << a << ", b = " << b << endl;
    }
};

int main() {
    int val = 20;
    Demo obj(10, val);
    obj.display();
    return 0;
}
```

Output:

```
Values initialized
a = 10, b = 20
```

`const` and `reference` members cannot be assigned inside constructor body —
so initialization list is **mandatory**.

Example 3: Initialization List with Inheritance

When a class is **derived**, you can use the initialization list to call the **base class constructor**.

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

class Base {
    int x;
public:
    Base(int a) {
        x = a;
        cout << "Base constructor called: x = " << x << endl;
    }
};

class Derived : public Base {
    int y;
public:
    Derived(int a, int b) : Base(a), y(b) {
        cout << "Derived constructor called: y = " << y << endl;
    }
};

int main() {
    Derived d(5, 10);
    return 0;
}
```

Output:

```
Base constructor called: x = 5
Derived constructor called: y = 10
```

The **Base class constructor** is called first using the initialization list,
then the **Derived constructor body** runs.

Example 4: With Virtual Base Class

In case of **virtual inheritance**,
the **most derived class** is responsible for initializing the **virtual base class**.

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

class A {
public:
    A(int x) { cout << "A constructor called with x = " << x << endl; }
};

class B : virtual public A {
public:
    B() : A(10) { cout << "B constructor\n"; }
};

class C : virtual public A {
public:
    C() : A(20) { cout << "C constructor\n"; }
};

class D : public B, public C {
public:
    D() : A(100) { cout << "D constructor\n"; }
};

int main() {
    D obj;
    return 0;
}
```

✅ **Output:**

```
A constructor called with x = 100
B constructor
C constructor
D constructor
```

Only the **most derived class (D)** initializes the **virtual base class (A)** —
even though B and C tried to call it.

Order of Initialization:

The **order of initialization** is determined by:

The **order of member declaration** in the class, **not** by the order in the initialization list.

```
class Example {
    int a;
    int b;
public:
    Example(int x, int y) : b(y), a(x) {
        cout << "a = " << a << ", b = " << b << endl;
    }
};
```

Even though `b(y)` comes before `a(x)`,
`a` is declared first, so it initializes first.