Inheritance in C++

Inheritance is a **feature of Object-Oriented Programming (OOP)** that allows one class to acquire the properties (data members) and behaviors (member functions) of another class.

It promotes **code reusability** and makes programs more **modular**.

Terminology

Base Class (Parent / Super Class) – The class whose properties and methods are inherited.

Derived Class (Child / Sub Class) – The class that inherits properties and methods from the base class.

Syntax

```
class Derived_class : access_specifier Base_class {
    // additional members or methods
};
```

Access specifiers in inheritance:

public – Base class `public` members become `public` in derived class; `protected` members remain `protected`.

protected – Base class `public` and `protected` members become `protected` in derived class.

private – Base class `public` and `protected` members become `private` in derived class.

Example

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

// Base class
class Vehicle {
public:
    string brand = "Toyota";
    void honk() {
        cout << "Tuut, tuut!" << endl;
    }
};

// Derived class
class Car : public Vehicle {   // public inheritance
public:
    string model = "Corolla";
};

int main() {
    Car myCar;

    // Accessing base class members
```

```
    cout << myCar.brand << endl;  // Toyota
    myCar.honk();                 // Tuut, tuut!

    // Accessing derived class members
    cout << myCar.model << endl;  // Corolla

    return 0;
}
```

Output:

```
Toyota
Tuut, tuut!
Corolla
```

Types of Inheritance

Type	Description
Single Inheritance	Derived class inherits from one base class.
Multiple Inheritance	Derived class inherits from more than one base class.
Multilevel Inheritance	A class is derived from another derived class.
Hierarchical Inheritance	Multiple derived classes inherit from a single base class.
Hybrid Inheritance	Combination of multiple and multilevel inheritance.

Key Points

- Inheritance promotes **code reusability**.
- Derived class can add new members or **override base class methods**.
- **Access specifiers** define how base class members are inherited.
- Multiple inheritance allows combining functionalities from multiple base classes.

Visibility (Access) Modes in Inheritance

Visibility mode determines how the **base class members** are accessible in the **derived class**.

Mode	Base class `public` members in derived	Base class `protected` members in derived	Base class `private` members in derived
public	public	protected	inaccessible
protected	protected	protected	inaccessible
private	private	private	inaccessible

Note: Private members of the base class are **never inherited** directly, but can be accessed via public/protected member functions of the base class.

Example with Visibility Modes

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

```
class Base {
public:
    int x = 10;
protected:
    int y = 20;
private:
    int z = 30;
};

// Public inheritance
class PublicDerived : public Base {
public:
    void show() {
        cout << "x = " << x << endl;  // Accessible
        cout << "y = " << y << endl;  // Accessible
        // cout << "z = " << z << endl; // Not accessible
    }
};

// Protected inheritance
class ProtectedDerived : protected Base {
public:
    void show() {
        cout << "x = " << x << endl;  // Accessible as protected
        cout << "y = " << y << endl;  // Accessible as protected
    }
};

// Private inheritance
class PrivateDerived : private Base {
public:
    void show() {
        cout << "x = " << x << endl;  // Accessible as private
        cout << "y = " << y << endl;  // Accessible as private
    }
};

int main() {
    PublicDerived pubObj;
    pubObj.show();
    cout << pubObj.x << endl;  // Accessible (public inheritance)

    ProtectedDerived protObj;
    protObj.show();
    // cout << protObj.x << endl; // Not accessible (protected inheritance)

    PrivateDerived privObj;
    privObj.show();
    // cout << privObj.x << endl; // Not accessible (private inheritance)

    return 0;
}
```

Output:

```
x = 10
y = 20
x = 10
y = 20
x = 10
```

```
y = 20
10
```

Single-Level Inheritance

Single-level inheritance is a type of inheritance in which a **derived class inherits directly from a single base class**.
It is the simplest form of inheritance where a child class gets properties and behaviors of one parent class.

Example with Variables `name`, `age` and optional `id`:

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

// Base class
class Person {
public:
    string name;
    int age;

    void showPerson() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

// Derived class
class Student : public Person {
public:
    int id;  // optional third variable

    void showStudent() {
        cout << "ID: " << id << ", Name: " << name << ", Age: " << age <<
endl;
    }
};

int main() {
    Student s1;
    s1.name = "Neeraj";
    s1.age = 23;
    s1.id = 101;

    s1.showPerson();   // Base class method
    s1.showStudent();  // Derived class method

    return 0;
}
```

Output:

```
Name: Neeraj, Age: 23
ID: 101, Name: Neeraj, Age: 23
```

Explanation:
The derived class `Student` inherits `name` and `age` from `Person`. It also has its own variable `id`.

Multi-Level Inheritance

Multi-level inheritance is a type of inheritance in which a **class is derived from another derived class**, forming a **chain of inheritance**.
It allows the last derived class to access members of all ancestor classes.

Example with Variables `name`, `age`, `id` and extra variable `marks`:

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

// Base class
class Person {
public:
    string name;
    int age;

    void showPerson() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

// Derived class 1
class Student : public Person {
public:
    int id;

    void showStudent() {
        cout << "ID: " << id << ", Name: " << name << ", Age: " << age <<
endl;
    }
};

// Derived class 2
class Result : public Student {
public:
    float marks;

    void showResult() {
        cout << "ID: " << id << ", Name: " << name << ", Age: " << age
             << ", Marks: " << marks << endl;
    }
};

int main() {
    Result r1;
    r1.name = "Neeraj";
    r1.age = 23;
    r1.id = 101;
    r1.marks = 95.5;

    r1.showPerson();   // Base class method
    r1.showStudent();  // Derived1 method
    r1.showResult();   // Derived2 method

    return 0;
}
```

Output:

```
Name: Neeraj, Age: 23
ID: 101, Name: Neeraj, Age: 23
ID: 101, Name: Neeraj, Age: 23, Marks: 95.5
```

Multiple Inheritance in C++

Multiple inheritance is a feature of C++ in which a **derived class inherits from more than one base class**.

It allows a class to **combine properties and behaviors** from multiple classes.

Syntax:

```
class Derived : access_specifier Base1, access_specifier Base2 {
    // additional members
};
```

Example:

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

class Student {
public:
    string name;
    void showName() {
        cout << "Name: " << name << endl;
    }
};

class Exam {
public:
    int marks;
    void showMarks() {
        cout << "Marks: " << marks << endl;
    }
};

// Derived class inherits from both Student and Exam
class Result : public Student, public Exam {
public:
    int id;
    void showResult() {
        cout << "ID: " << id << ", Name: " << name << ", Marks: " << marks
<< endl;
    }
};

int main() {
    Result r1;
    r1.name = "Neeraj";
    r1.marks = 95;
    r1.id = 101;

    r1.showResult(); // Access all members
```

```
    return 0;
}
```

Output:

```
ID: 101, Name: Neeraj, Marks: 95
```

Explanation: `Result` inherits properties and methods from **both** `Student` and `Exam`.

Ambiguity in Multiple Inheritance

If **two base classes have a member with the same name**, the derived class faces **ambiguity**, because the compiler cannot know which one to use.

Example of Ambiguity:

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

class Base1 {
public:
    void show() {
        cout << "Base1 show" << endl;
    }
};

class Base2 {
public:
    void show() {
        cout << "Base2 show" << endl;
    }
};

class Derived : public Base1, public Base2 {
};

int main() {
    Derived d;
    // d.show(); // ERROR: ambiguous
    return 0;
}
```

Error:

```
error: request for member 'show' is ambiguous
```

Ambiguity Resolution

To resolve ambiguity, we **specify the class name** using the **scope resolution operator** `::`.

```
int main() {
    Derived d;
    d.Base1::show();  // Calls Base1's show
    d.Base2::show();  // Calls Base2's show
    return 0;
}
```

Output:

```
Base1 show
Base2 show
```

Explanation: Using `Base1::show()` or `Base2::show()` tells the compiler **exactly which function to call**, resolving the ambiguity.

Key Points

- Multiple inheritance allows combining features from **more than one base class**.
- Ambiguity arises when **two base classes have members with the same name**.
- Ambiguity can be resolved using **scope resolution operator** `::`.
- Using **virtual base classes** can also help prevent **diamond problem** in complex hierarchies.