Inheritance

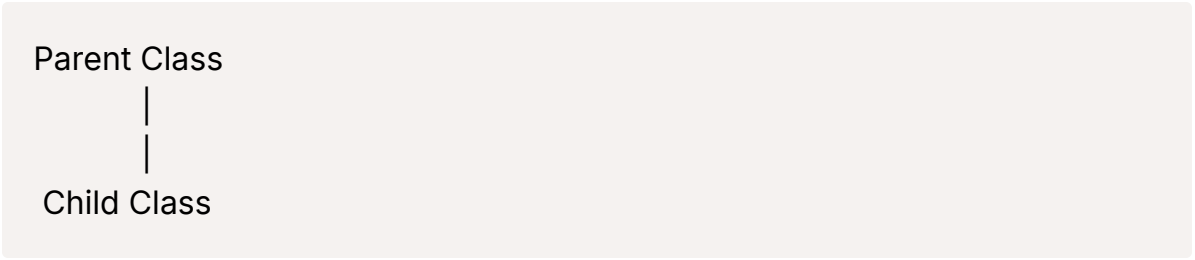
- The mechanism where one class (Derived Class) acquires features from another (Base Class), promoting code reusability.
- The **Base Class** is the existing class being inherited from, while the **Derived Class** is the new class that extends it.
- Members marked `protected` are private to the outside world but accessible to derived classes.

Example:

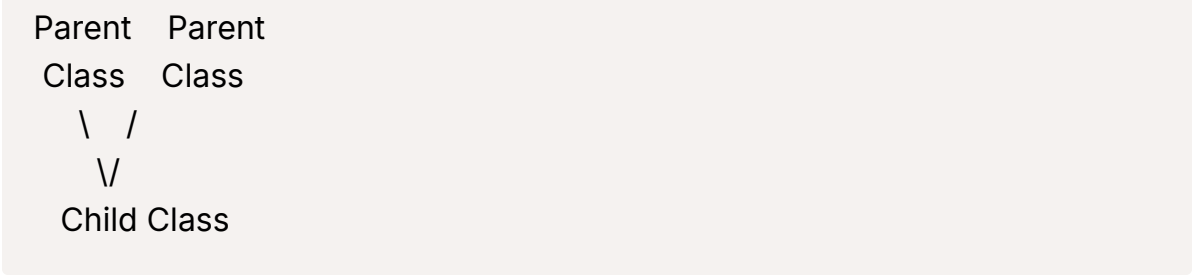
```
#include <iostream>
using namespace std;

class Parent {
protected:
    int id = 500; // Accessible to children
};

class Child : public Parent {
public:
    void showID() {
        cout << "Accessed protected ID: " << id << endl;
    }
};

int main() {
    Child obj;
    obj.showID();
    // cout << obj.id; // ERROR: id is protected
    return 0;
}
```

Types of Inheritance

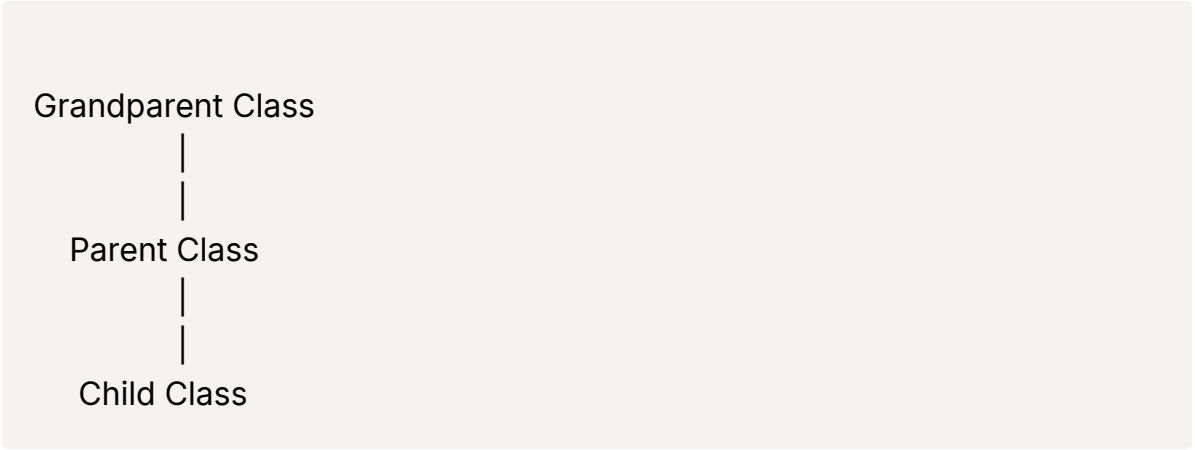
- **Single**: One derived class inherits from one base class.



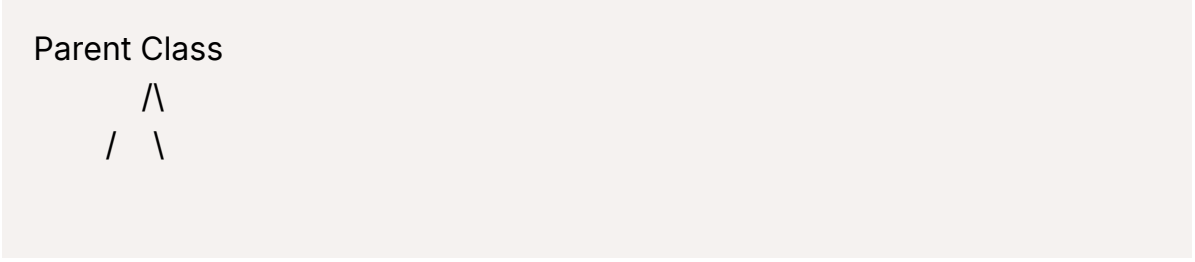
- **Multiple**: One derived class inherits from more than one base class.



- **Multilevel**: A derived class is inherited by another class, creating a chain (e.g., Vehicle → FourWheeler → Car).



- **Hierarchical**: Multiple derived classes inherit from a single base class.



```
 Child   Child
 Class      Class
```

- **Hybrid**: A combination of multiple and multilevel inheritance.

Example:

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

class Father { public: void height() { cout << "Tall" << endl; } };
class Mother { public: void eyes() { cout << "Blue" << endl; } };

// Multiple Inheritance
class Child : public Father, public Mother {};

int main() {
    Child c;
    c.height();
    c.eyes();
    return 0;
}
```

Diamond Problem

- Occurs in hybrid inheritance when a class receives two copies of a grandparent class's members through different paths, causing ambiguity.

```
 Grandparent Class
           /\
         /   \
  Parent  Parent
  Class   Class
     \    /
```

```
    \/
  Child Class
```

- Resolved by inheriting the grandparent class as `virtual`, ensuring only one instance is passed down.

Example:

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

class A { public: int val = 10; };

// Inherit as virtual to avoid duplicate 'val' in D
class B : virtual public A {};
class C : virtual public A {};

class D : public B, public C {};

int main() {
    D obj;
    cout << "Value: " << obj.val << endl; // No ambiguity thanks to 'virtual'
    return 0;
}
```

Constructors and Destructors in Inheritance

- **Constructor Order**: When a derived object is created, the base class constructor is called first, followed by the derived class constructor.
- **Multiple Inheritance Order**: Constructors are called in the order they are inherited.
- **Parameterized Constructors**: To call a base class parameterized constructor, it must be explicitly mentioned in the derived class's constructor initialization list.

- **Destructor Order**: Destructors are called in the exact opposite order of constructors (Derived first, then Base).

Example:

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

class Base {
public:
    Base() { cout << "Base Constructor" << endl; }
    ~Base() { cout << "Base Destructor" << endl; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived Constructor" << endl; }
    ~Derived() { cout << "Derived Destructor" << endl; }
};

int main() {
    Derived obj; // Watch the console output for the order
    return 0;
}
```