INHERITANCE

CHAPTER 6

TABLE OF CONTENTS

.....

Our content is
divided into the given topics.
Each part will be described in the slides
that follow

INHERITANCE

- Inheritance is the process of creating new classes, called derived classes, that inherit properties from existing or base classes.
- The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own.
- The base class is unchanged by this process.
- Inheritance is an essential part of object oriented programming. It provides code reusability.

Need for Inheritance

1. Capability of expressing the inheritance relationship that ensures closeness with the real
2. Allows reusability of code, i.e., addition of additional features to an existing class without modifying it.
3. Reusing existing code that has already been debugged saves time and money and increases a program’s reliability.

BASE AND DERIVED CLASS

A Derived class is the class which inherits the property of another class. The class from which the properties are inherited is known as base class. Derived class is also known as subclass or child class. Base class is also known as the parent class. The base class is unchanged by this inheritance process.

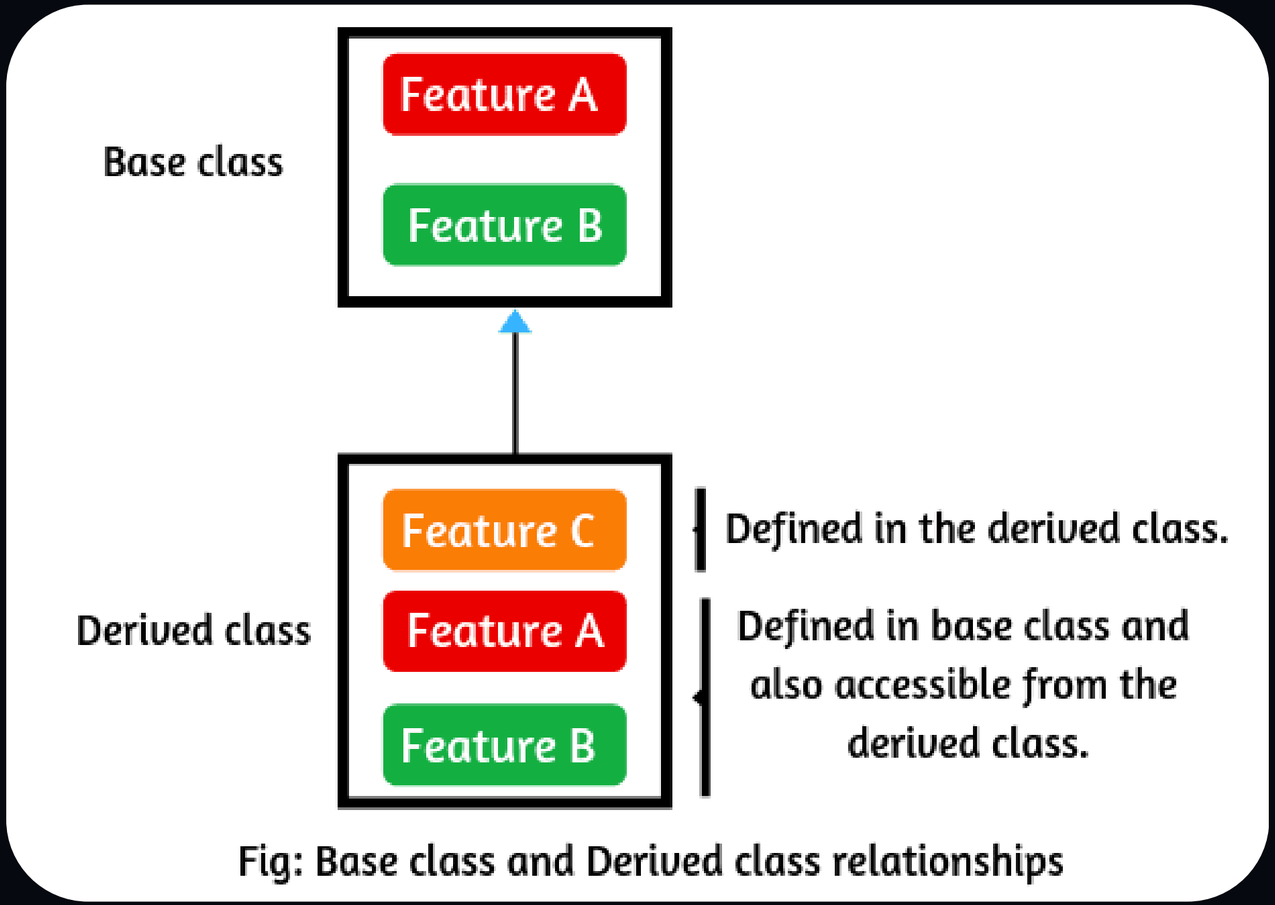


Fig: Base class and Derived class relationships

PROTECTED ACCESS SPECIFIER

Private access specifiers cannot be inherited and public members are inheritable but directly accessible through objects. Protected access specifier is used when the data members or member functions are required to be inheritable but inaccessible through objects. Like private members, they can be accessed only through functions.

So, If you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any data or functions that the derived classes might need to access should be made protected rather than private. This ensures that the class is “inheritance ready”.

DERIVED CLASS DECLARATION

```
class derived_class_name : visibility_mode base_class_name
{
    // members
};
```

The classname is followed by colon(:), visibility mode and base class name respectively. Visibility mode may be private, public, or protected.

Note: By default the visibility mode is private.

For inheritance from multiple classes, commas are used as shown below:

```
class derived_name : visibility base1_name, visibility base2_name
{
    // members
};
```

Visibility mode

Visibility Mode	Private Member of Base	Public Members of Base	Protected Members of Base
public	Not Inherited	public	protected
private	Not Inherited	private	private
protected	Not Inherited	protected	protected

- Private Members of Base class are never inherited.
- In public visibility mode, the public members are inherited as public members of derived class and protected members are inherited as protected members i.e. members are inherited in the same access specifier.
- Private visibility mode inherits both public and protected members in the private access section of the derived class.
- Protected visibility mode inherits both public and protected members into the protected section of the derived class.

```
/* single inheritance */
#include<iostream>
using namespace std;
class Base
{
    protected:
       int a;
    public:
       void inputBase()
       {
            cout<<"enter the value of a:";
            cin>>a;
       }
       void displayBase()
       {
           cout<<"the value of a is:"<< a;
           cout<<endl;
       }
};
```

```
class Derived: public Base
{
    int b;
    public:
    void inputDerived()
    {
        inputBase();
        cout<<"enter a value of b:";
        cin>>b;
    }
    void displayDerived()
    {
        displayBase();
        cout<<"the value of b is:"<< b;
        cout<<endl;
        cout <<"the sum ="<< a+b;
        cout<< endl;
    }
};
```

```
int main()
{
      Derived D1;
      D1.inputDerived();
      cout<<"Displaying the  taken value:";
      cout<<endl;
      D1.displayBase();
      D1.displayDerived();
      return 0;
}
```

Output:
enter the value of a:3
enter a value of b:5
Displaying the taken value:
the value of a is:3
the value of a is:3
the value of b is:5
the sum =8

MEMBER FUNCTION OVERRIDING

- The process of creating members in the derived class with the same name as that of the visible members of the base class is known as **function overriding.**
- It is called overriding because the new name in the derived class overrides (hides or displaces) the old name inherited from the base class.
- If we define a function on a derived class in the same name as an overloaded function in a base class, even with a different parameter list, the base class functions are hidden.
- C++ has a mechanism to access those functions; the base member can be accessed with the base class name and scope resolution operator before the function name.
- Redeclaration of member functions in derived class which is already defined inside visible sections (private and public) of base is known as **function overriding**.

In the next example, ‘Derived’ class is inherited from ‘Base1’ and ‘Base2’. All the classes have input() and display() functions. This is function overriding as the derived class has the same functions as base class. In main(), when input() and display() are invoked, the functions of the derived class are called.

Function Overriding

Note: The overridden functions of a base class can be invoked in two ways:

1. *From the member function of the derived class*
2. *From the object of derived class by using the scope resolution operator.(eg: obj.base::display();)*

```
#include<iostream>
using namespace std;
class Base1{
    protected:
        int a;
    public:
        void input(){
            cout<<"enter value to a of Base1: ";
            cin >> a;
        }
        void display(){
            cout <<"the a of Base 1 is:" << a << endl;
        }
};
```

```
class Base2
{
    protected:
        int a;
    public:
        void input(){
            cout<<"enter value of a for Base2: ";
            cin >> a;
        }
        void display(){
            cout<<"the a of Base2 is: " << a << endl;
        }
};
```

Function Overriding

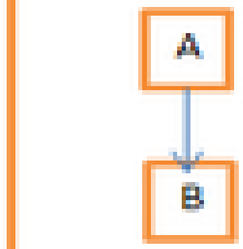
```
class Derived: public Base1, public Base2
{
    protected:
    int c;
    public:
        void input()
        {
            cout<<"enter the value of c: ";
            cin >> c;
        }
        void display()
        {
            Base1::display();
            Base2::display();
            cout<<"the value of c is:" << c << endl;
            cout<<"the sum is:" << Base1::a + c + Base2::a << endl;
        }
};
```

```
int main()
{
    Derived D1;
    D1.Base1::input();
    D1.Base2::input();
    D1.input();
    cout<<"displaying the taken value:" << endl;
    D1.display();
    return 0;
}
```

Output:

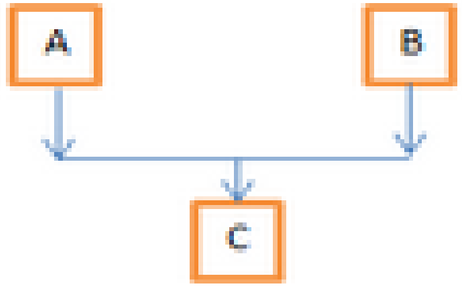
```
enter value to a of Base1: 3
enter value to a of Base2: 4
enter the value of c: 5
displaying the taken value:
the a of Base1 is:3
the a of Base2 is:4
the value of c is:5
the sum is:12
```

FORMS OF INHERITANCE

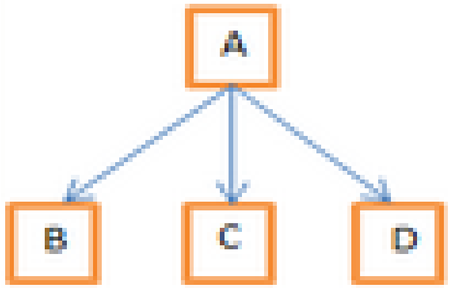


Single Inheritance

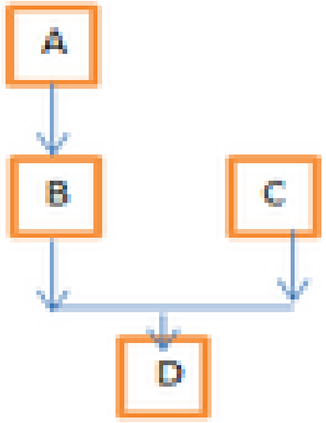
Multilevel Inheritance



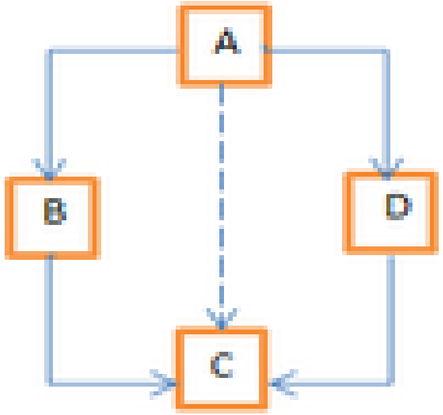
Multiple Inheritance



Hierarchical Inheritance



Hybrid Inheritance



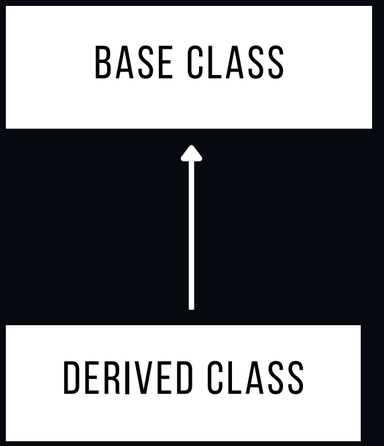
Multipath Inheritance

Single inheritance

This is the simplest form of inheritance. One derived class is inherited from one base class.

Syntax:

class derived_class_name: visibility_mode base_class_name
{ ……
};




```
#include <iostream.h>
class Value
{
protected:
int val;
public:
void set_values (int a){
val=a;
}
};
```

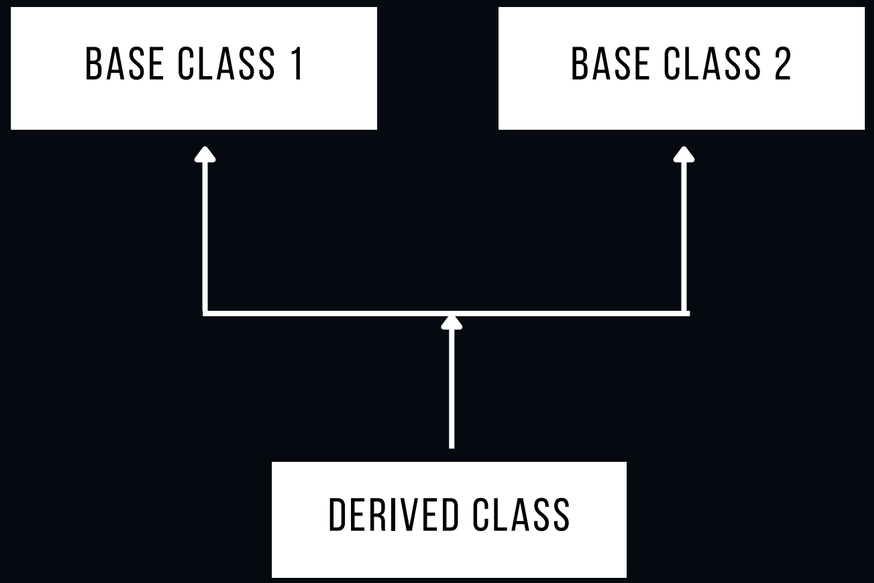
```
class Cube: public Value
{
public:
int cube()
{
    return (val*val*val);
}
};
```

```
int main ()
{
Cube cub;
cub.set_values (5);
cout << "The Cube of 5 is::" << cub.cube() <<
endl;
return 0;
}
```

Multiple inheritance

When a derived class is inherited from two or more base classes it is known as multiple inheritance.

```
class derived_class_name : visibility_mode base_class1, visibility_mode base_class2, ....
{
      ..............
};
```



Multiple inheritance

```
#include<iostream>
using namespace std;
class Base1
{
protected:
int a;
public:
void inputBase1()
{
cout<<" enter the value of a:" << endl;
cin >> a;
 }
void displayBase1()
{
cout<<"the value of a is:"<< a << endl;
}
};
class Base2
{
protected:
int b;
public:
void inputBase2()
{
cout<<"enter the value of b:" << endl;
cin >> b;
}
void displayBase2()
{
cout<<"the value of b is:" << b << endl;
}
};
```

Multiple inheritance

```
class Derived:public Base1, public Base2
{
int c;
public:
void inputDerived(){
cout<<"enter the value of c:" << endl;
cin >> c;
inputBase1();
inputBase2();
}
void displayDerived()
{
displayBase1();
displayBase2();
cout<<"the value of c is:" << c << endl;
cout<<"the sum is:" << a+b+c << endl;
}
};
```

```
int main()
{
Derived D1;
D1.inputDerived();
cout<<"displaying the taken value: "<< endl;
D1.displayDerived();
return 0;
}
```

Output:

```
enter the value of c: 3
enter the value of a: 1
enter the value of b: 2
displaying the taken value
the value of a is: 1
the value of b is: 2
the value of c is: 3
the sum is: 6
```

Ambiguity in multiple inheritance

When a derived class is inherited from multiple base classes,i.e., two or more base classes, and the base classes have the same functions, ambiguity arises. This ambiguity can be removed using the scope resolution operator (::).

In the next example, ‘Derived’ class is inherited from ‘Base1’ and ‘Base2’. Both ‘Base1’ and ‘Base2’ have the same functions input() and display(). When inherited into ‘Derived’, ambiguity arises as it is unclear whether the function of ‘Base1’ or ‘Base2’ is to be called. So, scope resolution is used to resolve this ambiguity.

Ambiguity in multiple inheritance

```
#include<iostream>
using namespace std;
class Base1
{
    protected:
        int x;
    public:
        void input()
        {
            cout<<"enter value to x of Base1: ";
            cin >> x;
        }
        void display()
        {
            cout <<"the x of Base 1 is:" << x << endl;
        }
};
```

```
class Base2
{
    protected:
        int x;
    public:
        void input()
        {
            cout<<"enter value of x for Base2: ";
            cin >> x;
        }
        void display()
        {
            cout<<"the x of Base2 is: " << x << endl;
        }
};
```

Ambiguity in multiple inheritance

```
class Derived: public Base1, public Base2
{
    protected:
    int c;
    public:
        void inputDerived(){
            cout<<"enter the value of c: ";
            cin >> c;
            Base1::input();
            Base2::input();
        }
        void displayDerived(){
            Base1::display();
            Base2::display();
            cout<<"the value of c is:" << c << endl;
            cout<<"the sum is:" << Base1::x + c + Base2::x << endl;
        }
};
```

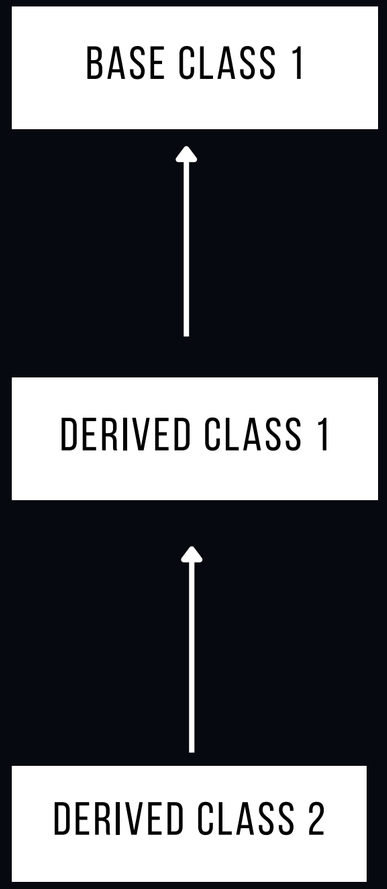
```
int main()
{
    Derived D1;
    D1.inputDerived();
    cout<<"displaying the taken value:" << endl;
    D1.displayDerived();
    return 0;
}
```

Output:

```
enter the value of c: 5
enter value to x of Base1: 4
enter value of x for Base2: 3
displaying the taken value:
the x of Base 1 is:4
the x of Base2 is: 3
the value of c is:5
the sum is:12
```

Multilevel inheritance

When a derived class acts as a base class for another class, it is known as multilevel inheritance.



Multilevel inheritance

```
#include <iostream>
using namespace std;
class A {
protected:
int a;
public:
void inputA() {
cout << "Enter the value of a:";
cin >> a;
}
void displayA() {
cout << "a = " << a << endl;
}
};
```

```
class B: public A{
protected:
int b;
public:
void inputB() {
inputA();
cout << "Enter the value of b:";
cin >> b;
}
void displayB() {
displayA();
cout << "b = " << b << endl;
}
};
```

Multilevel inheritance

```
class C: public B{
int c;
public:
void inputC() {
inputB();
cout << "Enter the value of c:";
cin >> c;
}
void displayC() {
displayB();
cout << "c = " << c << endl;
}
};
```

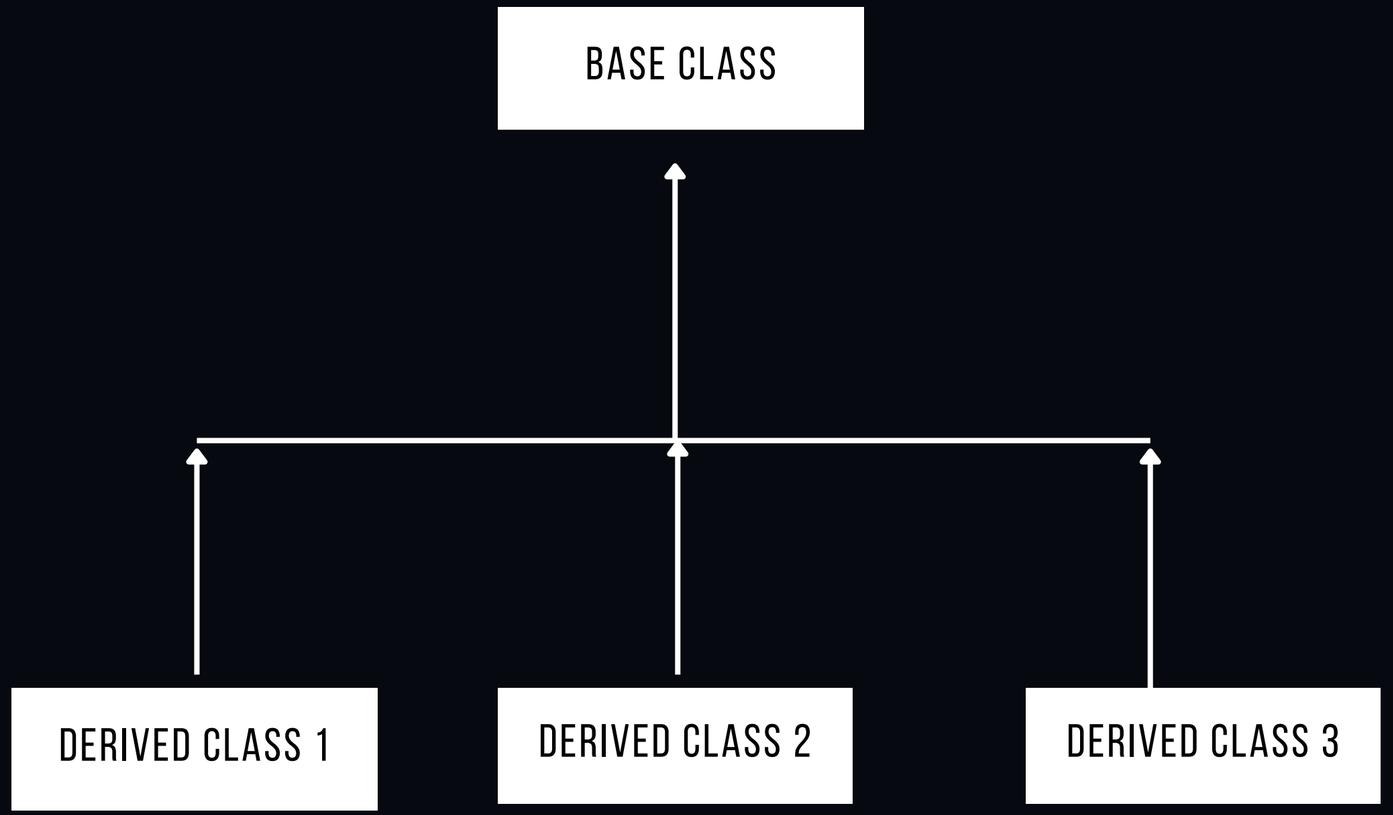
```
int main()
{
C obj;
obj.inputC();
obj.displayC();
return 0;
}
```

Output:

```
Enter the value of a:1
Enter the value of b:2
Enter the value of c:3
a = 1
b = 2
c = 3
```

Hierarchical inheritance

When two or more classes from one base class, it is known as hierarchical inheritance.



Hierarchical inheritance

```
#include <iostream>
using namespace std;
class A {
protected:
int a;
public:
void inputA() {
cout << "Enter the value of a:";
cin >> a;
}
void displayA() {
cout << "a = " << a << endl;
}
};
```

```
class B: public A{
protected:
int b;
public:
void inputB() {
inputA();
cout << "Enter the value of b:";
cin >> b;
}
void displayB() {
displayA();
cout << "b = " << b << endl;
}
};
```

Hierarchical inheritance

```
class C: public A{
int c;
public:
void inputC() {
inputA();
cout << "Enter the value of c:";
cin >> c;
}
void displayC() {
displayA();
cout << "c = " << c << endl;
}
};
```

```
int main()
{
cout << "Class B" << endl;
B obj1;
obj1.inputB();
obj1.displayB();
cout << "Class C" << endl;
C obj2;
obj2.inputC();
obj2.displayC();
return 0;
}
```

Output:

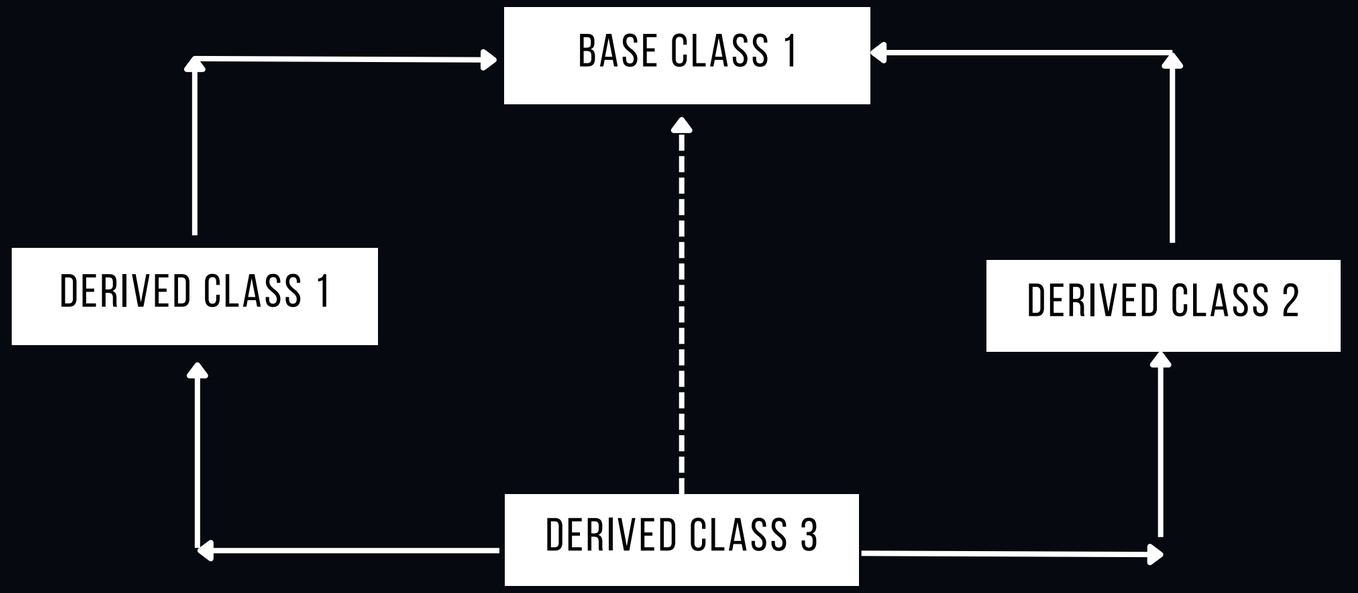
```
Class B
Enter the value of a:1
Enter the value of b:2
a = 1
b = 2
Class C
Enter the value of a:3
Enter the value of c:4
a = 3
c = 4
```

Hybrid inheritance

Hybrid Inheritance is a combination of more than one of the previous inheritance types(multiple, multilevel, hierarchical).

Multipath inheritance

When a base class is derived to two or more derived classes, and these derived classes are again combined as base class to another derived class, then this type of inheritance is known as multipath inheritance.



Multipath Inheritance and Virtual Base Class

Multipath inheritance can pose some problems in compilation. The public and protected members of grandparent are inherited into the child class twice, first, via parent 1 class and then via parent 2 class. Therefore, the child class would have duplicate sets of members of the grandparent which leads to ambiguity during compilation and it should be avoided.
It can be resolved adding virtual to the access specifier

```
class A //grandparent
{.....};
class B1 : virtual public A //parent 1
{.....};
class B2 : public virtual A //parent 2
{.....};
class C :public B1 , public B2 //child
{..... //only one copy of A will be inherited
.....
};
```

The keyword ***virtual*** and ***public*** or ***protected*** may be used in any order. After adding the keyword ***virtual*** while creating classes parent1 and parent2, it ensures that only one copy of the properties of class grandparent is inherited in the class child which is derived from classes parent1 and parent2.

Multipath Inheritance and Virtual Base Class

```
#include<iostream>
using namespace std;
class Aclass
{
protected:
int a;
public:
void inputA()
{
cout<<"enter the value of a: ";
cin >> a;
}
};
```

```
class Bclass: virtual public Aclass
{
protected:
int b;
public:
void inputB()
{
cout<<"enter value of b: " ;
cin >> b;
}
};
```

```
class Cclass:public virtual Aclass
{
protected:
int c;
public:
void inputC()
{
cout<<"enter value of c: ";
cin >> c;
}
};
```

Multipath Inheritance and Virtual Base Class

```
class Derived: public Bclass, public Cclass{
protected:
int d;
public:
void inputD(){
cout<<"enter value of d:" << endl;
cin >> d;
}
void display(){
cout<<"the value of a is:" << a << endl;
cout<<"the value of b is:" << b << endl;
cout<<"the value of c is:" << c << endl;
cout<<"the value of d is:" << d << endl;
cout<<"the sum is:" << (a+b+c+d) << endl;
}
};
```

```
int main(){
Derived D1;
D1.inputA();
D1.inputB();
D1.inputC();
D1.inputD();
D1.display();
return 0;
}
```

Output:

```
enter the value of a: 1
enter value of b: 3
enter value of c: 4
enter value of d: 5
the value of a is: 1
the value of b is: 3
the value of c is: 4
the value of d is: 5
the sum is: 13
```

Constructor Invocation in Single and Multiple Inheritances

/* constructor in simple inheritance */

```
#include<iostream>
using namespace std;
class Base1{
int x;
public:
Base1(){
cout<<"Base constructor" << endl;
}
~Base1(){
cout<<"Base destructor" << endl;
}
};
class Derived: public Base1{
int c;
public:
Derived(){
cout<<"Derived constructor" << endl;
}
~Derived(){
cout<<"Derived destructor" << endl;
}
};
int main()
{
Derived D1;
return 0;
}
```

Output:

Base constructor
Derived constructor
Derived destructor
Base destructor

Constructor Invocation in Single and Multiple Inheritances

/* access private of base class in derived class*/

```
#include<iostream>
using namespace std;
class Base1
{
    int x;
    public:
    void inputB(){
        cout<<"enter value to x of Base1:" << endl;
        cin >> x;
    }
    void display(){
        cout<<"the x of Base1 is:" << x;
    }
    int returnx(){
        return x;
}
};
```

```
class Derived: public Base1
{
    int c;
    public:
    void inputD()
    {
        cout<<"enter the value of c:" << endl;
        cin >> c;
     }
     void displayD()
     {
         display();
         cout<<" the value of c is:" << c << endl;
         cout<<" the sum is:" << returnx() + c << endl;
      }
};
```

Constructor Invocation in Single and Multiple Inheritances

```
int main()
{
    Derived D1;
    cout<<"the num of data element inside derived class:" <<sizeof(D1)/sizeof(int) << endl;
    D1.inputB();
    D1.inputD();
    cout<<"Displaying the taken value:" << endl;
    D1.displayD();
    return 0;
}
```

Output:

```
enter value to x of Base1:
5
enter the value of c:
3
Displaying the taken value:
the x of Base1 is:5 the value of c is:3
the sum is:8
```

Constructor Invocation in Single and Multiple Inheritances

Note: If a base class has a parameter constructor then there must be a constructor inside a derived class which passes a value to a base class parameters constructor through initialization list.

/*parameterized constructor through initialization list*/

```
#include<iostream>
using namespace std;
class Base1{
int a;
public:
Base1(int a){
cout<<"Base1 constructor:" << endl;
this ->a = a;
}
~Base1(){
cout<<" Base1 destructor:" << a << endl;
}
};
class Base2{
int c;
public:
Base2(int c){
this ->c = c;
cout<<"Base2 constructor:" << endl;
}
~Base2(){
cout<<"Base2 destructor:" << c << endl;
}
};
```

Constructor Invocation in Single and Multiple Inheritances

```
class Derived: public Base2, public Base1
{
      int d;
      public:
        Derived(): Base1(10), Base2(20)
        {
            d=0;
            cout<<"Derived constructor:" << endl;
        }
        Derived( int x, int y, int z): Base1(x), Base2(y)
        {
            d=z;
            cout<<"Derived constructor:" << endl;
        }
        ~Derived(){
            cout<<"Derived destructor:" << d << endl;
        }
};
```

```
int main()
{
      Derived D1;
      Derived D2(1,2,3);
      return 0;
}
```

Output:

```
Base2 constructor:
Base1 constructor:
Derived constructor:
Base2 constructor:
Base1 constructor:
Derived constructor:
Derived destructor:3
Base1 destructor:1
Base2 destructor:2
Derived destructor:0
Base1 destructor:10
Base2 destructor:20
```

Destructor Invocation in Single and Multiple Inheritances

Destructor in single inheritance

When an object of derived class is created, first the base class constructor is invoked, followed by the derived class constructors. When an object of derived class expires, first the derived class destructor is invoked, followed by the base class destructor.

Constructors and destructors of base class are not inherited by the derived class. Besides, whenever the derived class needs to invoke base class’s constructor or destructor, it can invoke them through explicitly calling them.

Destructor in single inheritance

```
#include<iostream>
using namespace std;
class base
{
 public:
 base()
 {
  cout<<"\nBase Class Constructor.";
 }
 ~base()
 {
  cout<<"\nBase Class Destructor.";
 }
};
class derived : public base
{
 public:
 derived()
 {
  cout<<"\nDerived Class Constructor.";
 }
 ~derived()
 {
  cout<<"\nDerived Class Destructor.";
 }
};
int main()
{
 derived D1;
 return 0;
}
```

Output:

Base constructor
Derived constructor
Derived destructor
Base destructor

Destructor Invocation in Single and Multiple Inheritances

Destructor in multiple inheritance

In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. The destructor execute in reverse order to that of constructor i.e., destructor of the derived class is called and then the destructor of the base class which is last, in declaration of derived class and followed by base class in reverse order.

Destructor in multiple inheritance

```
#include<iostream>
using namespace std;
class base1
{
 public:
 base1()
 {
  cout<<"\nBase1 Class Constructor.";
 }
 ~base1()
 {
  cout<<"\nBase1 Class Destructor.";
 }
};
```

```
class base2
{
 public:
 base2()
 {
  cout<<"\nBase2 Class Constructor.";
 }
 ~base2()
 {
  cout<<"\nBase2 Class Destructor.";
 }
};
```

Destructor in multiple inheritance

```
class derived : public base1, public base2
{
 public:
 derived()
 {
  cout<<”\nDerived Class Constructor.”;
 }
 ~derived()
 {
  cout<<”\nDerived Class Destructor.”;
 }
};
```

```
int main()
{
 Derived D1;
 return 0;
}
```

Output:

Base1 Class Constructor.
Base2 Class Constructor.
Derived Class Constructor.
Derived Class Destructor.
Base2 Class Destructor.
Base1 Class Destructor.

THANK YOU