Unit-2 Operator Overloading and Inheritance

2.1 Introduction to inheritance, Derived classes, Visibility Modes and Effects

2.2 Types of inheritance: Single, multilevel, multiple, hierarchical, Hybrid inheritance,

2.3 Overloading Unary and Binary Operators

2.1 Concepts of inheritance

- Inheritance is another important feature of Object Oriented Programming.
- Inheritance gives you the way to use already existing classes into the new class rather than writing existing class code again into new class.
- The mechanism of deriving a new class from an old class is called Inheritance.
- The old class is referred as base class and new class is referred as Derived class.
- The derived class inherits some or all the properties from the base class.

2.1.1 Defining Derived classes: -

Derived class can be defined by specifying its relationship with the base class.

Syntax: -

```
Class derivedclassname: VisibilityMode Baseclassname
{
        //code
}
```

In syntax the colon (:) indicates that the derived classname is derived from the base classname. The visibility mode specifies that in which way base class is derived.

2.1.2 Visibility Modes and Effects

The following table shows the visibility of base class derived into derived class.

Base class Visibility	**Derived Class Visibility (Visibility Modes)**		
	Private Derivation	**Public Derivation**	**Protected Derivation**
Private	Not Inherited	Not Inherited	Not Inherited
Public	Private	Public	Protected
Protected	Private	Protected	protected

```
Class A
{
private:
            int a;
protected:
            int Pa;
public:
            void getA();
            void putA();
};
Class B: public A       OR      class B: private A       OR       class B: protected A
{
private:
            int b;
protected:
            int Pb;
public:
            void getB();
            void putB();
};
```

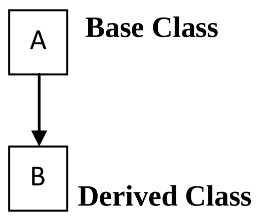
- When a base class is privately inherited by a derived class then public and protected member of base class becomes private member of derived class.
- When a base class is publicly inherited by a derived class then public member becomes public member of derived class and protected member of base class becomes protected member of derived class.
- When a base class is protected inherited by a derived class then public and protected member of base class becomes protected member of derived class.

Hence in all these three derivation private member of a base class is not inherited at all.

2.2 Types of inheritance

1) Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance
4) Hierarchical Inheritance
5) Hybrid Inheritance

1) Single Inheritance: -



Syntax: -

```
class A
{
	//code
};
class B:public A
{
	//code
};
```

In single inheritance one base class is present and only one derived class is present.

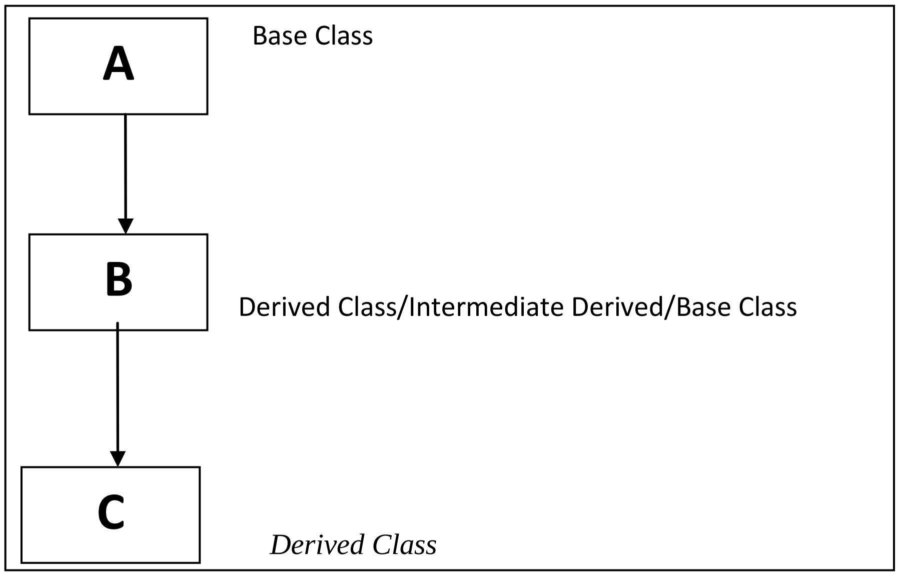
As in figure class B inherits all the properties of base class A.

As shown in Syntax Class B publicly derived from Class A so class B contains its own data and the data from the public section of class A.

// Program to Demonstrate use of Single Inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;
class A
{
private:
        int a;
public:
        void getA()
        {
         cout<<"Enter Value of a";
         cin>>a;
        }
        void putA()
        {
         cout<<"Value Of a="<<a;
        }
};
class B:public A
{
private:
        int b;
public:
        void getB()
        {
        cout<<"Enter Value of b";
        cin>>b;
        }
        void putB()
        {
        cout<<"Value Of b="<<b;
        }
};
int main()
{
B B1;
B1.getA();
B1.putA();
B1.getB();
B1.putB();
return(0);
}
```

2) Multilevel Inheritance-



Syntax:

```
        class A
        {
                code//;
                ______
        ______
    };
    clss B: public A
    {
        code;
        _________
        _________
    };
    class C :public B
    {
        code;
        ________
        ________
    };
```

When one Class is derived from another derived class then it is called as multilevel inheritance.

As in fig class A act as a base class for derived class B and derived class ‘B’ act as a base class for derived class ‘C’.

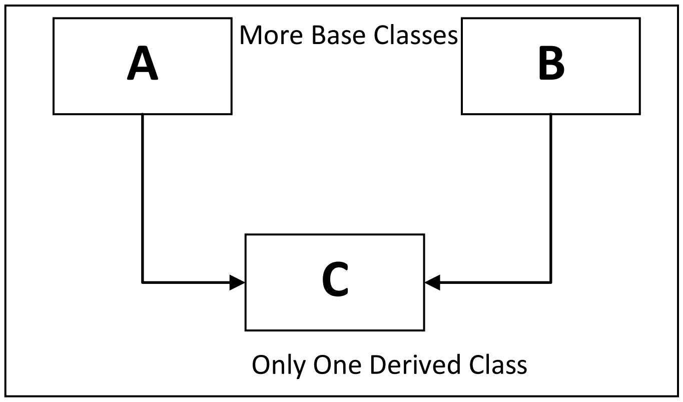
// Program to Demonstrate use of Multiplevel Inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;
class A
{
private:
    int a;
public:
    void getA()
    {
     cout<<"Enter Value of a";
     cin>>a;
```

```
        }
        void putA()
        {
         cout<<"Value Of a="<<a;
        }
};
class B:public A
{
private:
        int b;
public:
        void getB()
        {
        cout<<"Enter Value of b";
        cin>>b;
        }
        void putB()
        {
        cout<<"Value Of b="<<b;
        }
};
class C:public B
{
private:
        int c;
public:
        void getC()
        {
        cout<<"Enter Value of c";
        cin>>c;
        }
        void putC()
        {
        cout<<"Value Of c="<<c;
        }
};
int main()
{
C C1;
C1.getA();
C1.putA();
C1.getB();
C1.putB();
C1.getC();
C1.putC();
return(0);
}
```

In above program class C publicly inherits the details of class B and Class ‘B’ publicly derived from ‘A’, so we can call class ‘A’ and class ‘B’ public member functions by using object of class ‘C’.

3) Multiple Inheritance-



Syntax-

```
        class A
        {
         //code;
         _______
         _______
        };
        class B
        {
         //code;
         _______
         _______
        };
        class C : public A, public B
        {
         //code;
         _______
         _______
        };
```

When one class which is derived from more than one base classes then it is called as multiple inheritance.

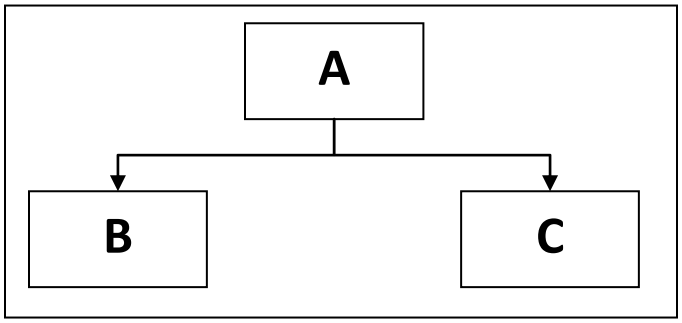
In multiple inheritance only one derived class and more than one base classes are present. Thus multiple inheritance allows us to combine the feature of several existing classes.

// Program to Demonstrate use of Multiple Inheritance

```
#include <iostream>
#include <conio.h>
using namespace std;
class A
{
private:
        int a;
```

```
public:
            void getA()
            {
             cout<<"Enter Value of a";
             cin>>a;
            }
            void putA()
            {
             cout<<"Value Of a="<<a;
            }
};
class B
{
private:
            int b;
public:
            void getB()
            {
            cout<<"Enter Value of b";
            cin>>b;
            }
            void putB()
            {
            cout<<"Value Of b="<<b<<"\n";
            }
};
class C:public A, public B
{
private:
            int c;
public:
            void getC()
            {
            cout<<"Enter Value of c";
            cin>>c;
            }
            void putC()
            {
            cout<<"Value Of c="<<c<<"\n";
            }
};
int main()
{
C C1;
C1.getA();
C1.putA();
C1.getB();
C1.putB();
C1.getC();
C1.putC();
return(0);
}
```

4) Hierarchical Inheritance-



Syntax-

```
class A
          {
           //code;
           _______
           _______
          };
class B : public A
          {
           //code;
           _______
           _______
          };
class C : public A
          {
           //code;
           _______
           _______
          };
```

It is a type of inheritance in which more than one classes derived from a single base class.

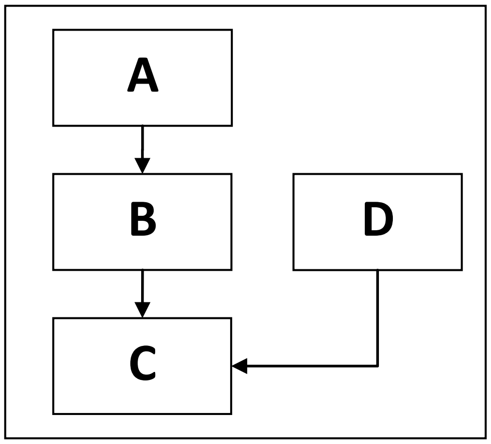
In hierarchical inheritance only one base class is present and more than one derived class is present.

All derived classes are derived from a common base class.

```
// Program to Demonstrate use of Hierarchical Inheritance
#include <iostream>
#include <conio.h>
using namespace std;
class A
{
private:
          int a;
```

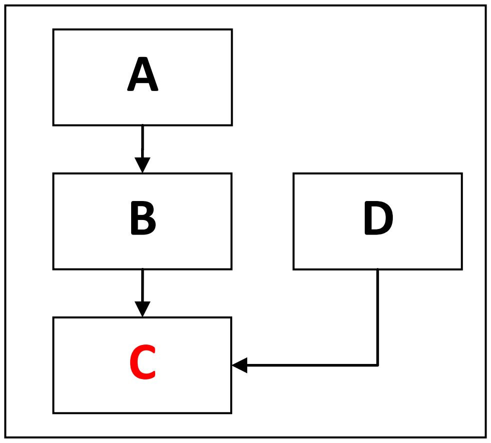
```
public:
            void getA()
            {
             cout<<"Enter Value of a";
             cin>>a;
            }
            void putA()
            {
             cout<<"Value Of a="<<a;
            }
};
class B: public A
{
private:
            int b;
public:
            void getB()
            {
            cout<<"Enter Value of b";
            cin>>b;
            }
            void putB()
            {
            cout<<"Value Of b="<<b<<"\n";
            }
};
class C:public A
{
private:
            int c;
public:
            void getC()
            {
            cout<<"Enter Value of c";
            cin>>c;
            }
            void putC()
            {
            cout<<"Value Of c="<<c<<"\n";
            }
};
int main()
{
B B1;
B1.getA();
B1.putA();
B1.getB();
B1.putB();
C C1;
C1.getA();
C1.putA();
C1.getC();
C1.putC();
return(0);
}
```

5) Hybrid Inheritance :-



Syntax-

```
class A
{
        //code;
        _______
        _______
};
class B : public A
{
        //code;
        _______
        _______
};
class D
{
        //code;
        _______
        _______
};
class c : public B,public D
{
        //code;
        _______
        _______
};
```



Hybrid inheritance is a combination of multiple and multilevel inheritance as in figure the class C will have both they multilevel multiple inheritance. Thus they class C has features of all they other classes.

```
class A
{
        int a;
```

```
 public:
void geta()
{
              cout<<"Enter the value of a";
              cin>>a;
}
void puta()
{
              cout<<a;
}
};
class B : public A
{
             int b;
public:
void getb()
{
              cout<<"Enter value of b";
              cin>>b;
}
void putb()
{
              cout<<b;
}
};
class D
{
             int D;
 public:
 void getd()
 {
        cout<<"Enter value of d";
        cin>>d;
}
void  putd()
{
        cout<<d;
}
};
class C : public B,public D
{
       int c;
public:
void getc()
{
        cout<<"Enter value of c";
        cin>>c;
}
void putc()
{
        cout<<c;
}
};
void main()
{
       C C1;
       C1.geta();
       C1.puta();
       C1.getb();
       C1.putb();
       C1.getc();
       C1.putc();
```

```
        C1.getd();
        C1.putd();
        getch();
}
```

2.3 Operator Overloading

Operator overloading is the mechanism to give special meaning to an operator for the different data type.

We can give the special meaning to an operator but we are not going to change the basic meaning of an operator.

To overload the operator we have a special operator function **operator()** in C++.

Syntax of declaring operator Function inside class: -

```
        <return type> operator  op(argument list);
```

Defining operator Function outside class: -

```
<return type> <class name>:: operator op(arguments list)
{
        //code
}
```

- Here return type is the type of value return by the specified operation in the operator ().
- Operator is the keyword which tells to the compiler that it is a operator function.
- OP is the operator being overloaded.

Operator () function must be either member function or friend function but not the static function of a class.

By using this operator function we can overload 2 types of operators:-

1) Unary Operator : ++,-- (Requires only one operand) Ex: a++
2) Binary Operator :+,-,*,/ (Requires Two Operands) Ex: a + b

Rules for operator overloading: -

1) Only existing operators can be overloaded.
2) New operators cannot be created.
3) We cannot change the basic meaning of an operator.
4) The overloaded operator must have at least one operand.
5) Overloaded operator follows the syntax & rules of original operators.
6) Unary and binary operators overloaded using member function as well as friend function of a class.
7) Following operators cannot be overloaded-

   1] pointer to member (, .*)

   2] Scope Resolution Operator (::)

   3) Conditional Operator (?)

   4) Size Of Operator (sizeof())

   5) Membership Operator (.)

Function	**No of arguments required**	
	Unary	**Binary**
Member Function	No arguments	One explicit argument (01)
Friend Function	One reference argument (01)	Two explicit operator (02)

1. Unary operator Overloading:-

- The unary operators operate on a single operand and following are the examples of Unary operators −
  - The increment (++) and decrement (--) operators.
  - The unary minus (-) operator.

    - The logical not (!) operator.

- The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.
- Following example explain how minus (-) operator can be overloaded for prefix as well as postfix usage

```
Class data

{

int x,y,z;

public:

void accept()

{

cout<<”Enter x,y,z value”;

cin>>x>>y>>z;

}

void display()

{

cout<<x<<y<<z;

}

void operator -();

};

void data :: operator –()

{
        x = -x;
        y = -y;
        z= -z;
}
void main()
{
Data D1;
D1.accept();
-D1;                    //Call Operator –() Function
D1.display();
getch();
}
```

2. Binary Operator Overloading

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, + is a binary operator that works on the operands num and 9.

When we overload the binary operator for user-defined types by using the code:

```
obj3 = obj1 + obj2;
```

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

```
class Distance
```

```
{
public:
    // Member Object
    int feet, inch;
    // No Parameter Constructor
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }

    // Constructor to initialize the object's value
    // Parametrized Constructor
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }

    // Overloading (+) operator to perform addition of
    // two distance object
    Distance operator+(Distance& d2) // Call by reference
    {
        // Create an object to return
        Distance d3;

        // Perform addition of feet and inches
        d3.feet = this->feet + d2.feet;
        d3.inch = this->inch + d2.inch;

        // Return the resulting object
        return d3;
    }
};

// Driver Code
int main()
{
    // Declaring and Initializing first object
    Distance d1(8, 9);

    // Declaring and Initializing second object
    Distance d2(10, 2);

    // Declaring third object
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;
    // Display the result
    cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
    return 0;
}
```