UNIT-III

P. Madhuravani

UNIT – III

Operator Overloading: Overloading unary, binary operators, data conversion, pitfalls of operators overloading and conversion keywords, Explicit and Mutable.

Inheritance: Derived class and Base class, Single Inheritance, Multiple Inheritance, Multilevel Inheritance, Hierarchical Inheritance, Virtual Base Classes, Abstract Classes, Constructor in Derived Classes.

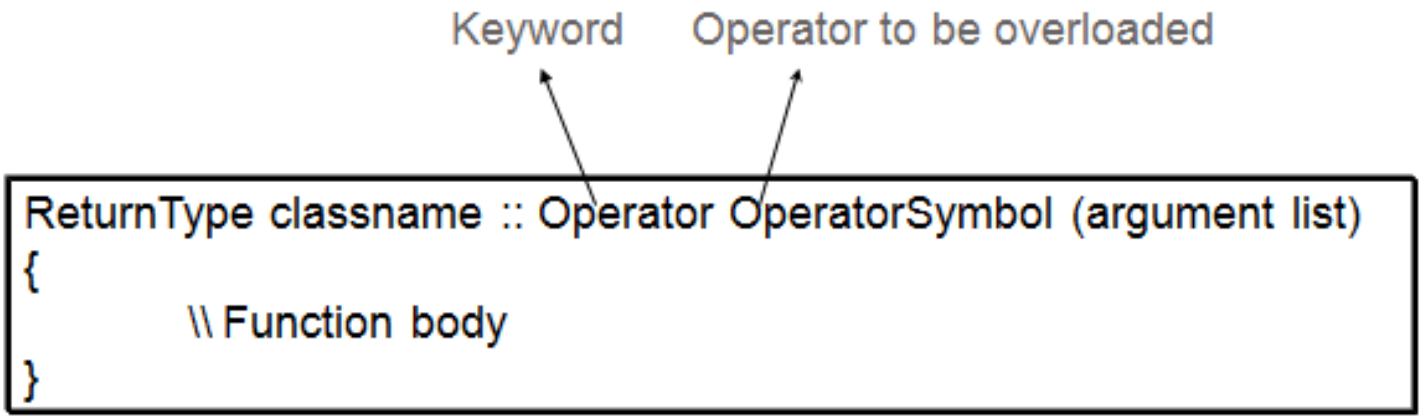
OPERATOR OVERLOADING

C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

The process of using same operator for different purpose is called operator overloading.

To define an additional task to an operator, specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called **operator function.**

General Form:



The process of overloading involves following steps:

- Create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op() in the public part of the class. It may be a member function or a friend function.
- Define the operator function to implement the required operations.

Rules for overloading operator:

•Only existing operators can be overloaded. New operators cannot be overloaded.

•Overloaded operator have at least one operand that is of user-defined type.

•When operator is overloaded we cannot change the basic meaning of an operator.
Ex: (+) operator cannot be used to subtract one value from the other.

•Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.

•C++ operators that can be overloaded are:

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

•There are some operators that cannot be overloaded.

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

•We cannot use friend functions to overload certain operators. However, member functions can be used to overload them.

Where friend cannot be used			
= Assignment operator	() Function call operator	[] Subscripting operator	-> Class member access operator

•When unary operator is overloaded by member function it has no explicit arguments and return no explicit values, but those overloaded by friend function, take one reference argument.

•Binary operators overloaded through a member function take one explicit argument and which are overloaded through a friend function take two explicit argument.

•When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

• Binary arithmetic operators such as +,-, * and / must explicitly return a value.

Overloaded operator functions can be invoked by expressions such as

Operator Function	Unary Operators	Binary Operators
Member Functions	op x or x op	x op y (or) x.operator op(y)
Friend Functions	operator op(x)	operator op(x,y)

Overloading Unary Operators

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--.

Overloading Unary Operators

Operator Function as Member Function

```
#include<iostream.h>
class space
{
        int x,y,z;
        public:
        void getdata(int a,int b,int c);
        void display();
       void operator-();
};
void space :: getdata(int a,int bmint c)
{
        x=a;
        y=b;
        z=c;
}
void space :: display()
{
      cout<<x<<y<<z;
}
void space :: operator-()
{
        x=-x;
        y=-y;
        z=-z;
}
void main()
{
        space S;
       S.getdata(10,-20,30);
       S.dispaly();
        -S;
       S.display();
}
```

Output:

10 -20 30

-10 20 -30

Operator Function as Friend function

```
#include<iostream.h>
class space
{
        int x,y,z;
        public:
            void getdata(int a,int b,int c);
            void display();
             friend void operator-(space);
};
void space :: getdata(int a,int b,int c)
{
      x=a;
      y=b;
      z=c;
}
void space :: display()
{
      cout<<x<<y<<z;
}
void operator-(space s)
{
       s.x=-s.x;
       s.y=-s.y;
        s.z=-s.z;
}
void main()
{
      space S;
      S.getdata(10,-20,30);
      S.dispaly();
     operator-(S);
     S.display();
}
```

Output:

10 -20 30

-10 20 -30

Following example explain how minus (-) operator can be overloaded

```
#include <iostream>

class Distance
{
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // method to display distance
    void displayDistance()
    {
      cout << "F: " << feet << " I:" << inches <<endl;
    }
// overloaded minus (-) operator
    Distance operator- ()
    {
      feet = -feet;
      inches = -inches;
      return Distance(feet, inches);
    }
};
int main()
{
  Distance D1(11, 10), D2(-5, 11);

  -D1;                  // apply negation
  D1.displayDistance();  // display D1

  -D2;                  // apply negation
  D2.displayDistance();  // display D2

  return 0;
}
```

Output:

F: -11 I:-10

F: 5 I:-11

```
/* C++ program to demonstrate the working of
++ operator overlading. */
#include <iostream>
class Check
{       private:
                int i;
        public:
                Check()
                 {
                        i=0;
                }
                Check operator ++()
                {
                        Check temp;
                        ++i; /* i increased by 1. */
                        temp.i=i;
                        return temp;
                }
                void Display()
                {
                        cout<<"i="<<i<<endl;
                }
    };
```

```
int main()
{
        Check obj, obj1;
        obj.Display();
        obj1.Display();
        obj1=++obj;
        obj.Display();
        obj1.Display();
        return 0;
}
```

Output:

i=0
i=0
i=1
i=1

Overloading Binary Operators

Operator Function as Member Function

```
#include<iostream.h>
class complex
{
float x,y;
public:
     complex(){ }
     complex(float real, float imag)
     {
              x=real;
              y=imag;
      }
     complex operator+(complex);
     void display();
};
complex complex::operator+(complex c)
{
        complex temp;
        temp.x=x+c.x;
        temp.y=y+c.y;
        return temp;
}
void complex::display()
{
          cout<<x<<"i"<<y;
}
main()
{
             complex c1,c2,c3;
             c1=complex(2.5,3.5);
            c2=complex(1.6,2.7);
            c3=c1+c2;
             c3.display();
}
```

Operator Function as Friend function

```
 #include<iostream.h>
class complex
{
float x,y;
public:
          complex(){ }
          complex(float real, float imag)
         {
                   x=real;
                  y=imag;
         }
         friend complex operator+(complex, complex);
          void display();
};
complex operator+(complex a, complex b)
{
           return complex((a.x+b.x),(a.y+b.y));
}
void complex::display()
{
          cout<<x<<"i"<<y;
}
main()
{
           complex c1,c2,c3;
           c1=complex(2.5,3.5);
           c2=complex(1.6,2.7);
          c3=operator+(c1,c2);
          c3.display();
}
```

overload the assignment operator (=)

```
#include <iostream>
class Distance
{
        private: int feet;
        Distance()
       {
            feet = 0;
            inches = 0;
       }
       Distance(int f, int i)
       {
            feet = f;
            inches = i;
       }
       void operator=(const Distance &D )
       {
              feet = D.feet;
              inches = D.inches;
       }
      void displayDistance()
      { cout << "F: " << feet << " I:" << inches <<
 endl; } };
```

```
int main()
{
        Distance D1(11, 10), D2(5, 11);
        cout << "First Distance : ";
        D1.displayDistance();
        cout << "Second Distance :";
        D2.displayDistance();
        // use assignment operator
        D1 = D2;
        cout << "First Distance :";
        D1.displayDistance();
        return 0;
}
```

Output:

First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11

```
//Overload ( ) operator
#include <iostream>
using namespace std;

class Distance
{
  private:
    int feet;             // 0 to infinite
    int inches;           // 0 to 12
  public:
    // required constructors
    Distance(){
      feet = 0;
      inches = 0;
    }
    Distance(int f, int i){
      feet = f;
      inches = i;
    }
    // overload function call
    Distance operator()(int a, int b, int c)
    {
      Distance D;
      // just put random calculation
      D.feet = a + c + 10;
      D.inches = b + c + 100 ;
      return D;
    }
    // method to display distance
    void displayDistance()
    {
      cout << "F: " << feet <<  " I:" <<  inches << endl;
    }
};
int main()
{
  Distance D1(11, 10), D2;

  cout << "First Distance : ";
  D1.displayDistance();

  D2 = D1(10, 10, 10); // invoke operator()
  cout << "Second Distance :";
  D2.displayDistance();

  return 0;
}
```

Output:

First Distance : F: 11 I:10
Second Distance :F: 30 I:120

```
//Overload [ ] opearator
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay
{
  private:
    int arr[SIZE];
  public:
    safearay()
    {
      register int i;
      for(i = 0; i < SIZE; i++)
      {
        arr[i] = i;
      }
    }
    int &operator[](int i)
    {
       if( i > SIZE )
       {
          cout << "Index out of bounds" <<endl;
          // return first element.
          return arr[0];
       }
       return arr[i];
    }
};
```

```
int main()
{
  safearay A;

  cout << "Value of A[2] : " << A[2] <<endl;
  cout << "Value of A[5] : " << A[5]<<endl;
  cout << "Value of A[12] : " << A[12]<<endl;

  return 0;
}
```

Output:

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

Data Conversions

- The type conversions are automatic only when the data types involved are built-in types.

```
int m;
float x = 3.14159;
m = x; //  convert x to integer before its value is assigned
       //  to m.
```

- For user defined data types, the compiler does not support automatic type conversions.
- We must design the conversion routines by ourselves.

Data Conversions

Different situations of data conversion between incompatible types.

- Conversion from basic type to class type.
- Conversion from class type to basic type.
- Conversion from one class type to another class type.

Basic to Class Type

The conversion from basic type to class type is easy to accomplish.

A constructor to build a string type object from a char * type variable.

```
string : : string(char *a)
{
  length = strlen(a);
  P = new char[length+1];
  strcpy(P,a);
}
```

The variables length and p are data members of the class string.

Basic to Class Type

```
string s1, s2;
string name1 = "IBM PC";
string name2 = "Apple Computers";
s1 = string(name1);
s2 = name2;
```

First converts name1 from char* type to string type and then assigns the string type value to the object s1.

First converts name2 from char* type to string type and then assigns the string type value to the object s2.

Basic to Class Type

```
class time
{      int hrs ;
       int mins ;
   public :
       …
   time (int t)      //constructor
   {
       hrs = t / 60 ;    //t in minutes
       mins = t % 60;
   }
} ;

time T1;  //object T1 created
int duration = 85;
T1 = duration;  // int to class type
```

After this conversion, the hrs member of T1 will contain a value of 1 and mins a value of 25, denoting 1 hours and 25 mins.

The constructors used for the type conversion take a single argument whose type is to be converted.

```
class data
{
    int x;
    float f;
    public :
    data()
    {
    x=0;
    f=0;
    }
    data ( float m)
    {
    x=2;
    f=m;
    }
    void show()
    {
    cout<<"\n x= "<<x<<" f="<<f;
    cout<<"\n x= "<<x<<"f="<<f;
    }
};
```

```
int main()
{
    clrscr();
    data z;
    z=1;
    z.show();
    z=2.5;
    z.show();
    return 0;
}
```

OUTPUT

x= 2 f = 1
x= 2 f = 1
x= 2 f = 2.5
x= 2 f = 2.5

Class To Basic Type

A constructor function do not support type conversion from a class type to a basic type.

An overloaded ***casting operator*** is used to convert a class type data to a basic type.

It is also referred to as ***conversion function***.

```
operator typename( )
{
   …
   …  ( function statements )
   …
}
```

This function converts a ***calss type*** data to ***typename***.

Class To Basic Type

```
vector : : operator double( )
{
  double sum = 0;
  for (int i=0; i < size ; i++)
    sum = sum + v[i] * v[i];
  return sqrt (sum);
}
```

This function converts a vector to the square root of the sum of squares of its components.

The operator double() can be used as follows:

```
double length = double(V1);  //V1 is an object of type vecto
```

Or

```
double length = V1;
```

Class To Basic Type

The casting operator function should satisfy the following conditions:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

```
vector : : operator double( )
{
  double sum = 0;
  for (int i=0; i < size ; i++)
    sum = sum + v[i] * v[i];
  return sqrt (sum);
}
```

Class To Basic Type

- Conversion functions are member functions and it is invoked with objects.
- Therefore the values used for conversion inside the function belong to the object that invoked the function.
- This means that the function does not need an argument.

```
#include<iostream.h>
#include<conio.h>

class data
{
   int x;
   float f;
   public:
   data()
   {
   x=0;
   f=0;
   }
   operator int()
   {
   return (x);
   }
   operator float()
   { return f; }
   data ( float m)
   {
   x=2;
   f=m;
   }
   void show()
   {
   cout<<"\n x= "<<x<<"f="<<f;
   cout<<"\n x= "<<x<<"f="<<f;
   }
};
```

```
int main()
{
   clrscr();
   int j;
   float f;
   data a;
   a=5.5;
   j=a; // operator int() is executed
   f=a; // operator float() is executed
   cout<<"\n Value of j :"<<j;
   cout<<"\n Value of f :"<<f;
   return 0;
}
```

OUTPUT

Value of j : 2
Value of f : 5.5

One Class To Another Class Type

objX = objY ; // objects of different types

- **objX** is an object of class **X** and **objY** is an object of class **Y**.
- The **class Y** type data is converted to the **class X** type data an the converted value is assigned to the **objX**.
- Conversion is takes place from **class Y** to **class X**.
- **Y** is known as ***source class***.
- **X** is known as ***destination class***.

One Class To Another Class Type

Conversion between objects of different classes can be carried out by either a constructor or a conversion function.

Choosing of constructor or the conversion function depends upon where we want the type-conversion function to be located in the source class or in the destination class.

One Class To Another Class Type

operator typename( )

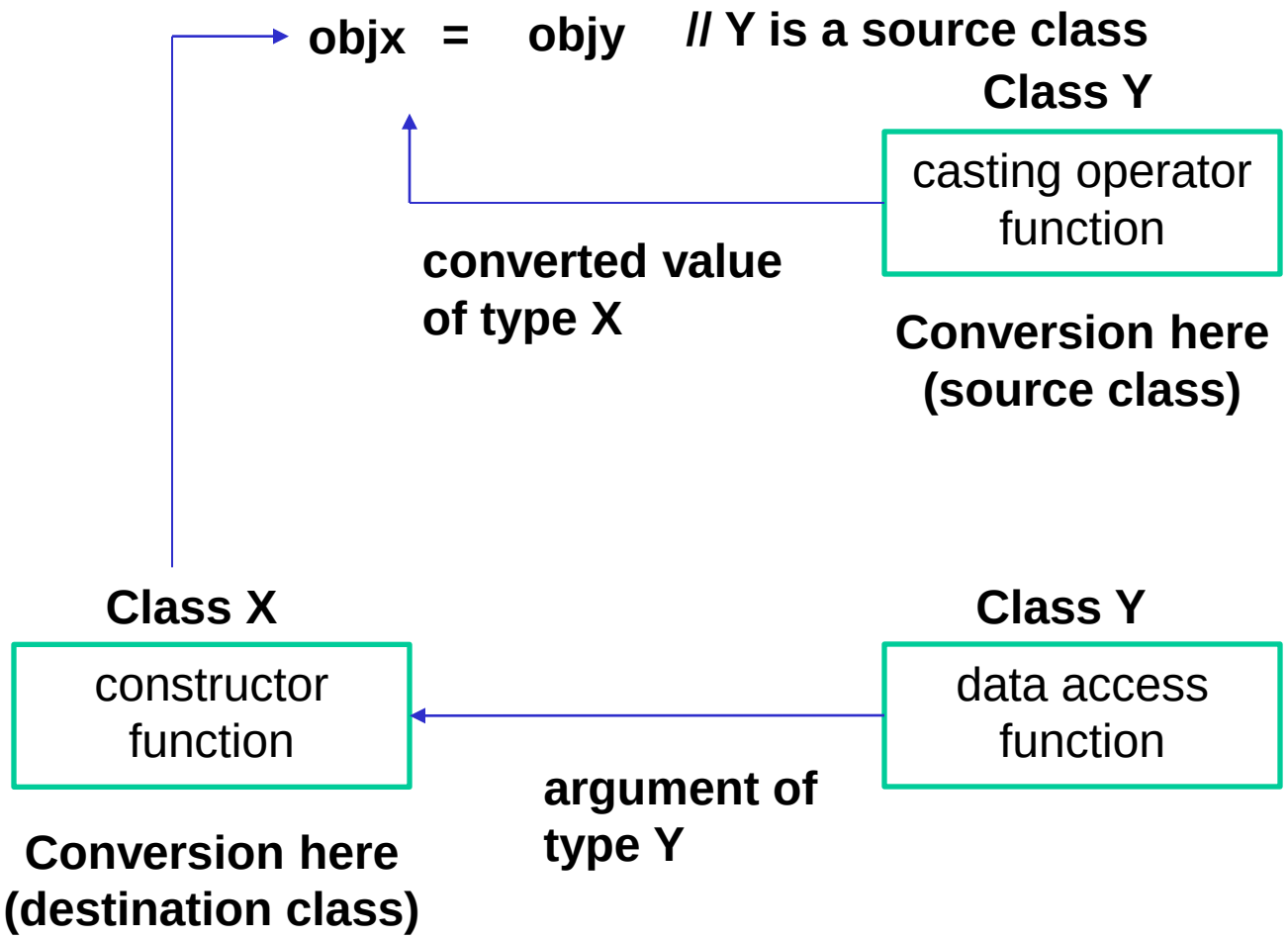
- Converts the class object of which it is a member to typename.
- The typename may be a built-in type or a user-defined one.
- In the case of conversions between objects, typename refers to the destination class.
- When a class needs to be converted, a casting operator function can be used at the source class.
- The conversion takes place in the source class and the result is given to the destination class object.

One Class To Another Class Type

Consider a constructor function with a single argument

- Construction function will be a member of the destination class.
- The argument belongs to the source class and is passed to the destination class for conversion.
- The conversion constructor be placed in the destination class.

One Class To Another Class Type



Conversion between object

```
#include<iostream.h>
#include<conio.h>

class minutes
{
   int m;
   public:
   minutes()
   { m=240; }
   get()
   { return (m);}
   void show()
   { cout<<"\n Minutes="<<m; }
};
class hours
{
   int h;
   public:
   void operator = (minutes x);
   void show()
   { cout<<"\n Hours="<<h; }
};
```

```
void hours:: operator = (minutes x)
{
   h=x.get()/60;
}
int main()
{
   clrscr();
   minutes minute;
   hours hour;
   hour=minute;
   minute.show();
   hour.show();
   return 0;
}
```

OUTPUT

Minutes = 240

Hours = 4

Data Conversions

Table provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

Conversion required	**Conversion takes place in**	
	Source Class	Destination Class
Basic -> class	Not applicable	Constructor
Class -> basic	Casting operator	Not applicable
Class - >class	Casting operator	Constructor

pitfalls of operators overloading and conversion keywords

Because the compiler must choose how to quietly perform atype conversion, it can get into trouble if you don't design yourconversions correctly. A simple and obvious situation occurs with aclass X that can convert itself to an object of class Y with anoperator Y( ) . If class Y has a constructor that takes a singleargument of type X, this represents the identical type conversion.

The compiler now has two ways to go from X to Y, so it willgenerate an ambiguity error when that conversion occurs:

pitfalls of operators overloading and conversion keywords

```
// Ambiguity in type conversion
class Y; // Class declaration
class X
{
public:
operator Y() const; // Convert X to Y
};
class Y
{
public:
Y(X); // Convert X to Y
};
```

```
int main()
{
    X x;
    Y y;
    x=y // Error: ambiguous
conversion
} ///:~
```

The obvious solution to this problem is not to do it: Just provide a single path for automatic conversion from one type to another.

explicit

The explicit keyword is used to declare class constructors to be "explicit" constructors.

Any constructor called with one argument performs implicit conversion in which the type received by the constructor is converted to an object of the class in which the constructor is defined. Since the conversion is automatic, we need not apply any casting.

In case, no such automatic conversion to take place, we may do so by declaring the one-argument constructor as **explicit**:

explicit

```
class ABC
{
int m;
public:
    explicit ABC(int i)
    {
        m=i;
    }
    ……..
};
```

Hear, objects of ABC class can be created using only the following form:
ABC abc(100);

The automatic conversion form
ABC abc1 = 100;
is not allowed and illegal. This is allowed when keyword explicit is not applied to the conversion.

What is explicit keyword?

explicit

- A constructor that takes a single argument operates as an implicit conversion operator by default. This is also referred as converting constructor.
- To prevent this implicit conversion keyword explicit has been introduced in C++. This makes the constructor as nonconverting.
- This keyword has effect only when defined on a single argument constructor or on a constructor which has default arguments for all but one argument.
- Compile time error will be reported if an attempt is made to create an object via conversion as in statements like

" MyClass obj2 = 200;".

mutable

class object or a member function may be declared as **const** thus making their member data not modifiable.

A situation may arise where we want to create a const object (or function) but we would like to modify a particular data item only. In such situations we can make that particular data item modifiable by declaring the item as **mutable**.

mutable

```
#include<iostream.h>
class ABC
{
    private:
        mutable int m;
    public:
        explicit ABC(int x=0)
        {
            m = x;
        }
        void change() const  //const function
        {
            m=m+10;
        }
        int display() const //const function
        {
            return m;
        }
int main()
{
    const ABC abc(100);
    cout<<"abc contains:"<<abc.display();
    abc.change();
cout<<"\n abc now contains: "<<abc.display();
    return 0;
}
```

Output:
abc contains : 100
abc now contains : 110

Note: Although the function change() declared constant, the value of m has been modified. Try to execute after deleting **mutable**.

Derived Class and Base Class

- Reusability is an important feature of OOP.
- C++ strongly supports the concept of reusability.
- The mechanism of deriving a new class from an old one is called inheritance (or derivation).
- The old class is referred to as **base class**.
- The new class is called the **derived class** or subclass.
- The derived class inherits some or all of the traits from the base class.
- A class can also inherit properties from more than one class

Derived Class and Base Class

- In inheritance, some of the base class data elements and member functions are inherited into the derived class.
- We can add our own data and member functions for extending the functionality of the base class.
- It is a powerful tool for incremental program development.
- Can increase the capabilities of an existing class without modifying it.

Derived Class and Base Class

A derived class can be defined by specifying its relationship with the base class in addition to its own details.

```
class derived-class-name : visibility-mode base-class-name
{
    ..........//
    ..........//    members of derived class
    ..........//
};
```

Derived Class and Base Class

class derived-class-name : visibility-mode base-class-name

The colon indicates that the derived-class-name is derived from the base-class-name

```
{
      ……..//
      ……..//    members of derived class
      ……..//
};
```

The visibility mode is optional and , if present, may be either private or public.

The default visibility mode is private.

Visibility mode specifies whether the features of the base class are derived privately or publicly.

Derived Class and Base Class

- When a base class is privately derived by a derived class, “public members” of the base class become “private members” of the derived class.
- Therefore the members of the derived class can only access the public members of the base class.
- They are inaccessible to the objects of the derived class.
- No member of the base class is accessible to the objects of the derived class.
- When a base class is publicly inherited, ”public members” of the base class become the “public members” of the derived class.
- They are accessible to the objects of the derived class.

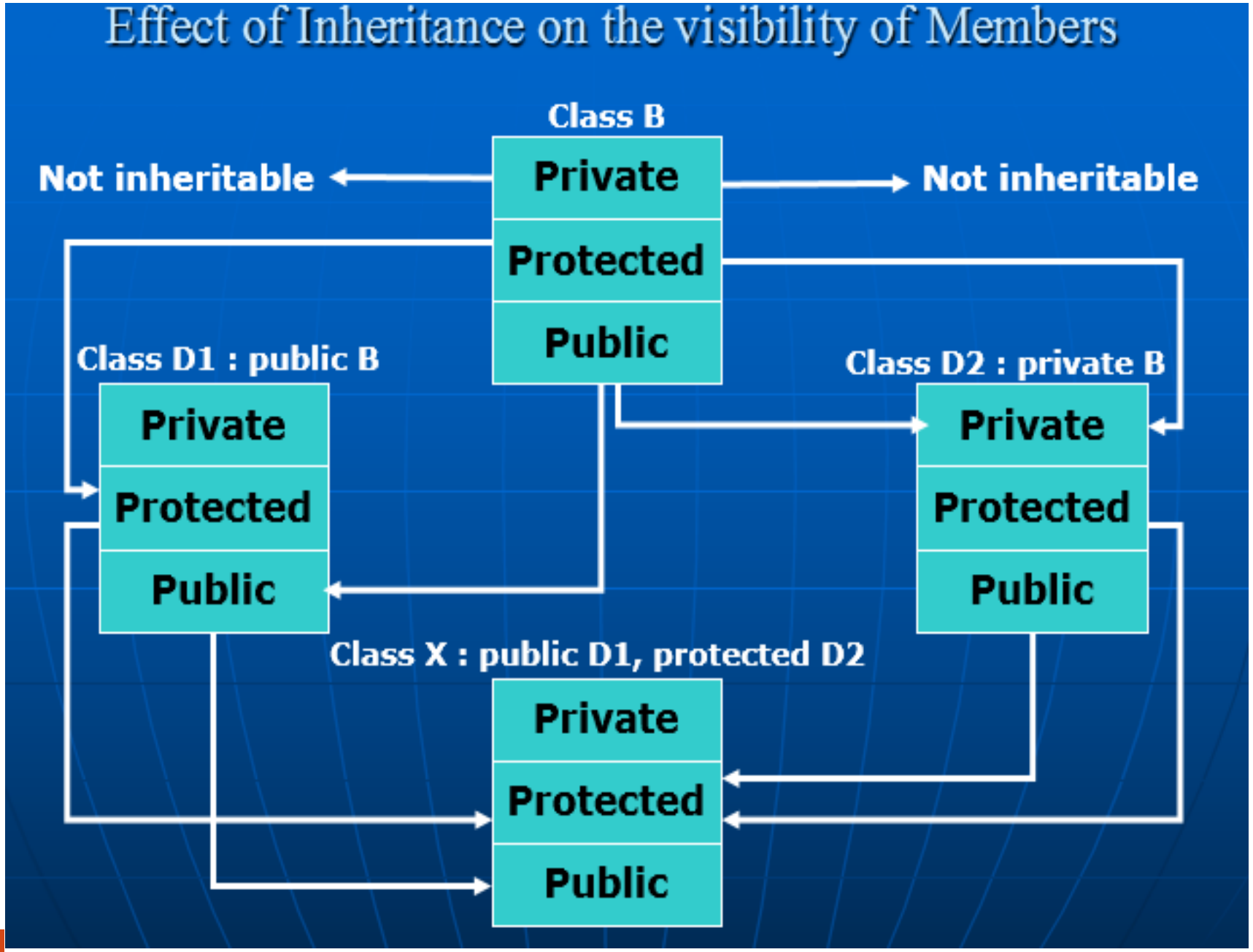
Derived Class and Base Class

- The private members of the base class are not inherited in both the cases (publicly/privately inherited).
- The private members of a base class will never become the members of its derived class.
- It is possible to inherit a base class in protected mode – **protected derivation.**
- In protected derivation, both the public and protected members of the base class become protected members of the derived class.

Derived Class and Base Class

- When a protected member is inherited in public mode, it becomes protected in the derived class.
- They are accessible by the member functions of the derived class.
- And they are ready for further inheritance.

- When a protected member is inherited in private mode, it becomes private in the derived class.
- They are accessible by the member functions of the derived class.
- But, they are not available for further inheritance.

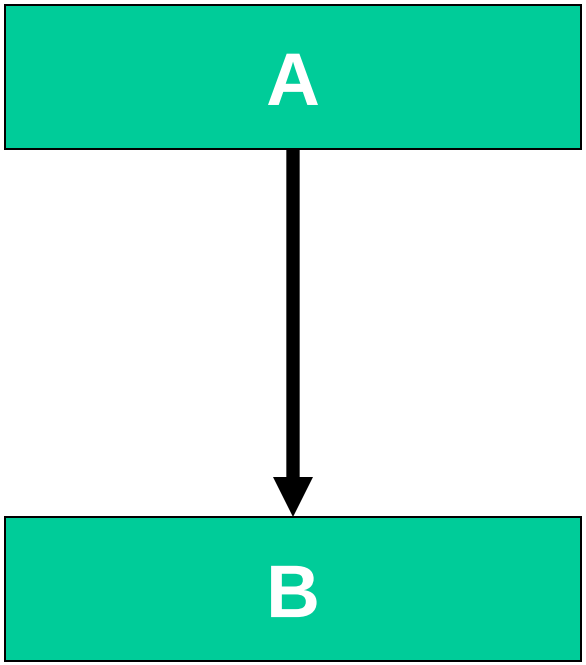

Effect of Inheritance on the visibility of Members
Class B
Not inheritable
Private
Not inheritable
Protected
Public
Class D1 : public B
Class D2 : private B
Private
Private
Protected
Protected
Public
Public
Class X : public D1, protected D2
Private
Protected
Public

Derived Class and Base Class

Base class visibility	Derived class visibility		
	Public Derivation	Private Derivation	Protected Derivation
Private →	Not Inherited	Not Inherited	Not Inherited
Protected →	Protected	Private	Protected
Public →	Public	Private	Protected

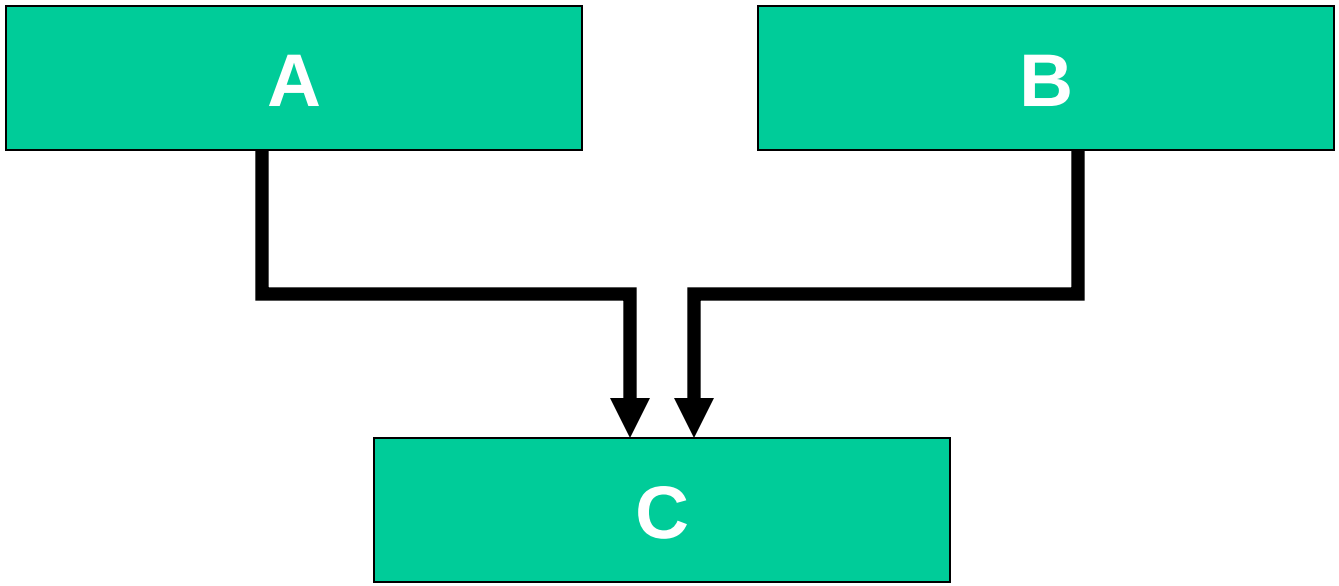
Single Inheritance

- A derived class with only one base class.



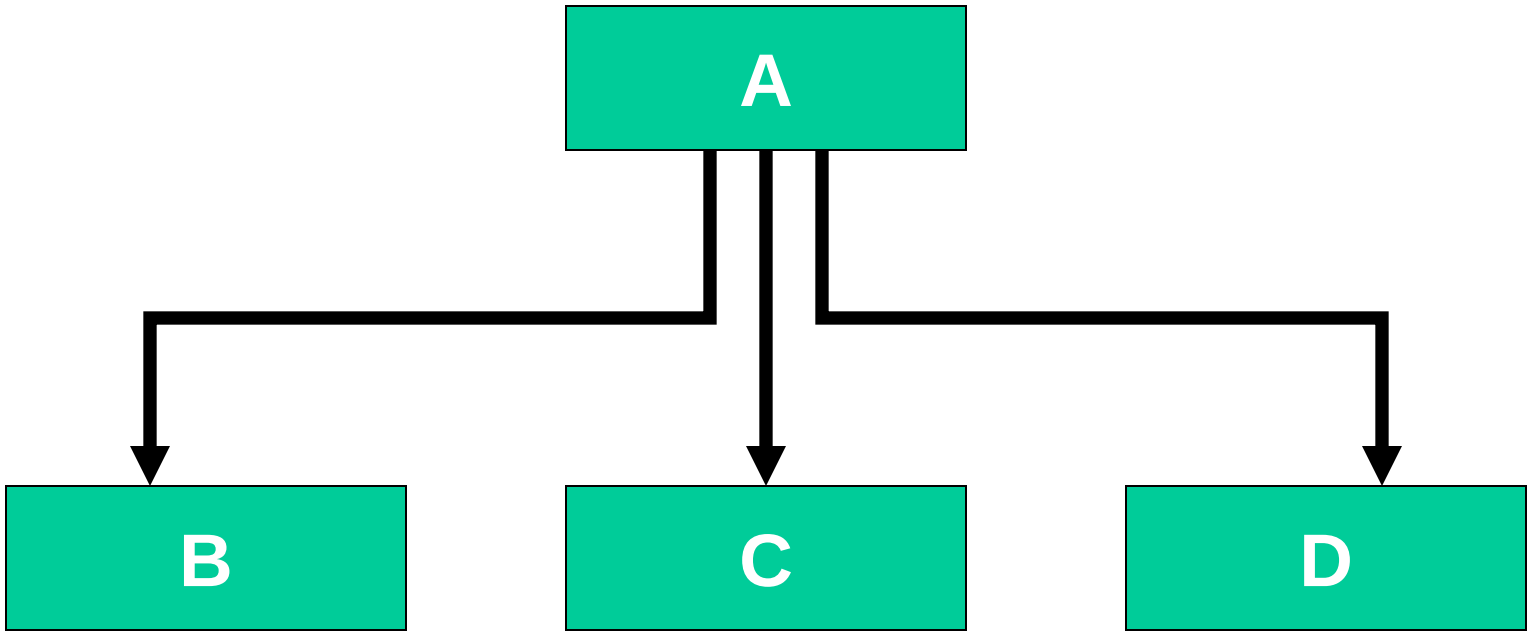
Multiple Inheritance

- A derived class with several base classes.



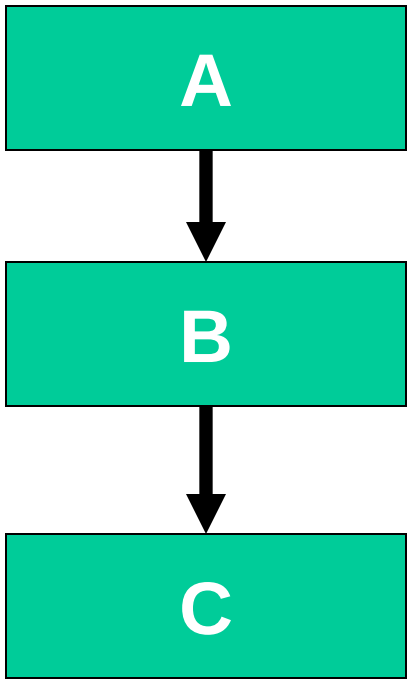
Hierarchical Inheritance

- A traits of one class may be inherited by more than one class.



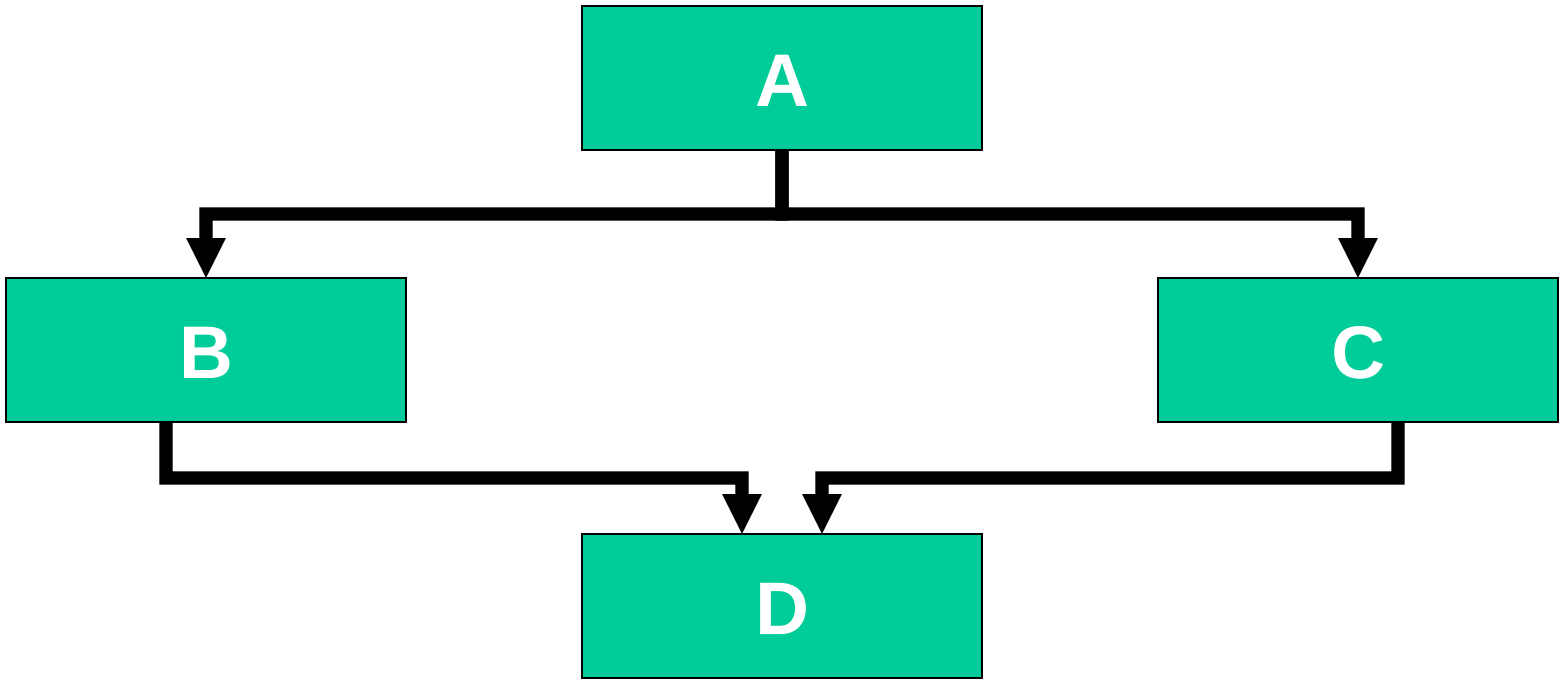
Multilevel Inheritance

- The mechanism of deriving a class from another derived class.



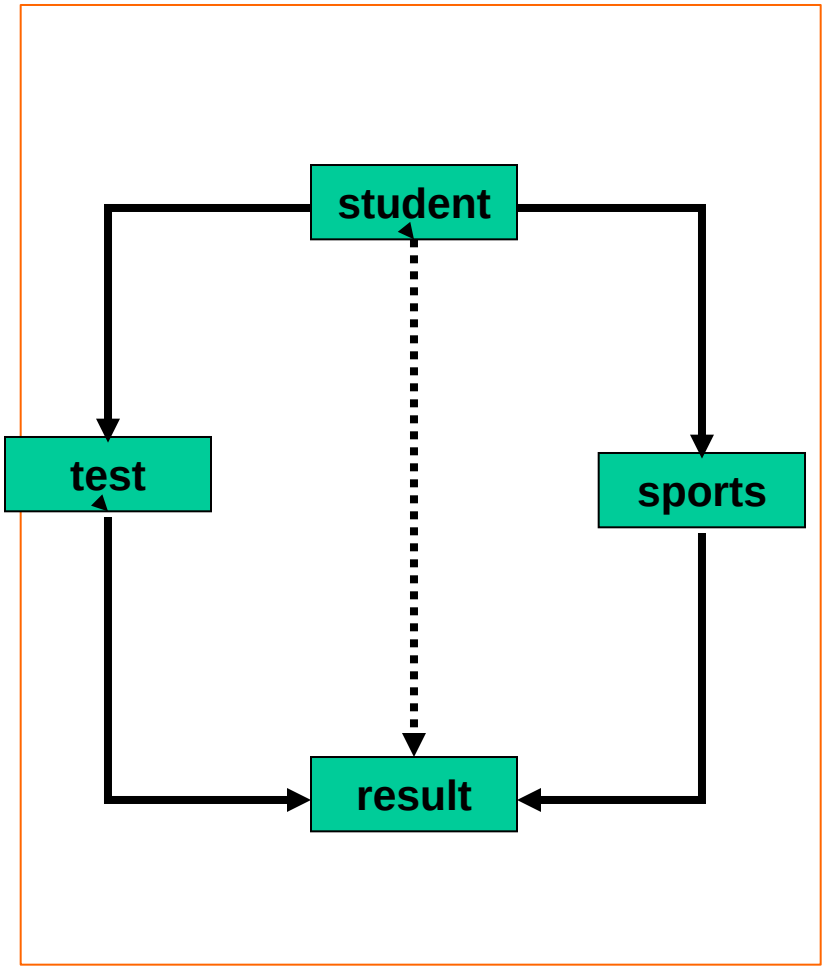
Hybrid Inheritance

- The mechanism of deriving a class by using a mixture of different methods.



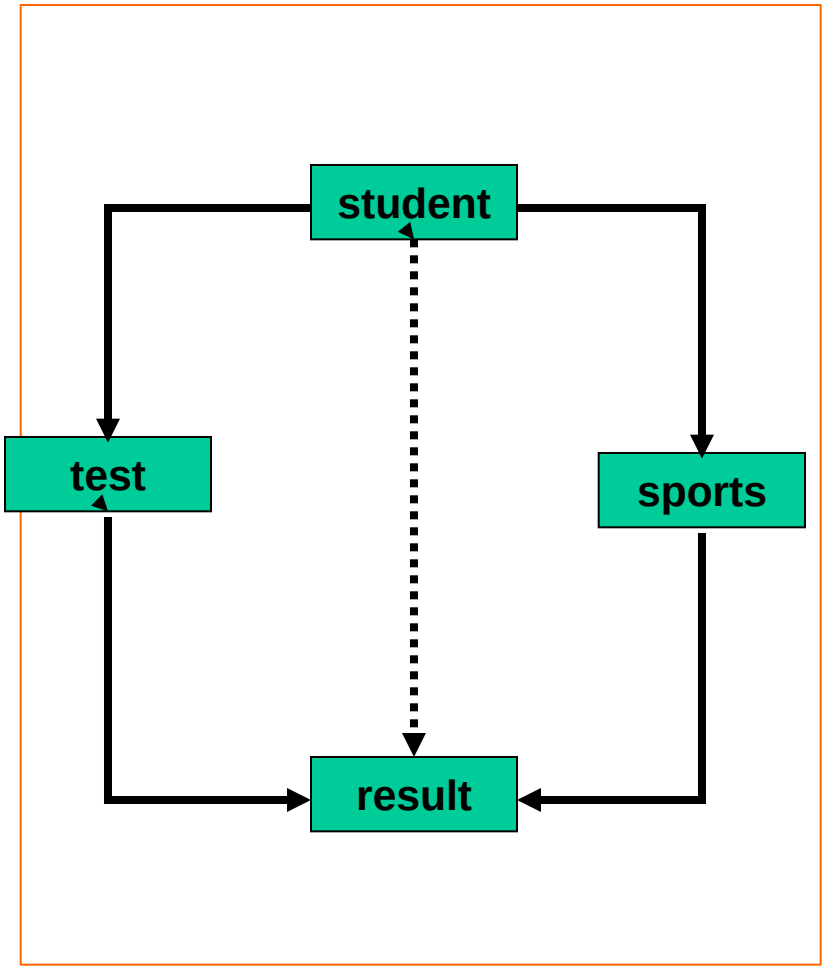
Virtual Base Classes

- Here the result class has two direct base classes test and sports which themselves have a common base class student.
- The result inherits the traits of student via two separate paths.



Virtual Base Classes

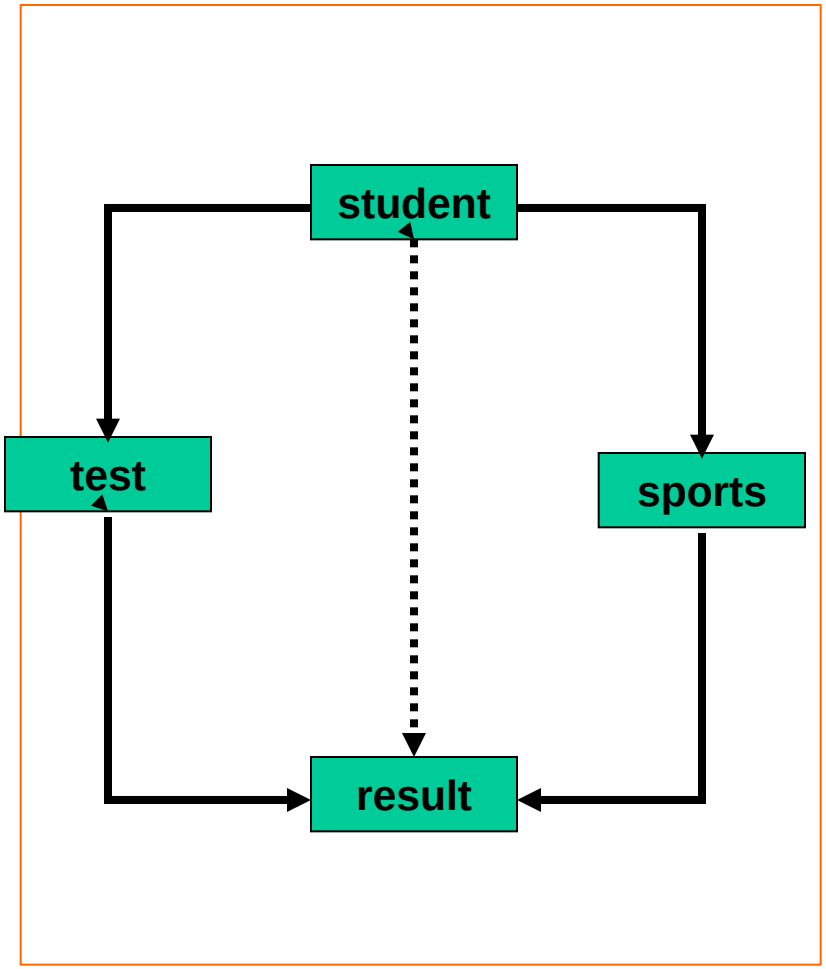
- It can also inherit directly as shown by the broken line.
- The student class is referred to as indirect base class.



Virtual Base Classes

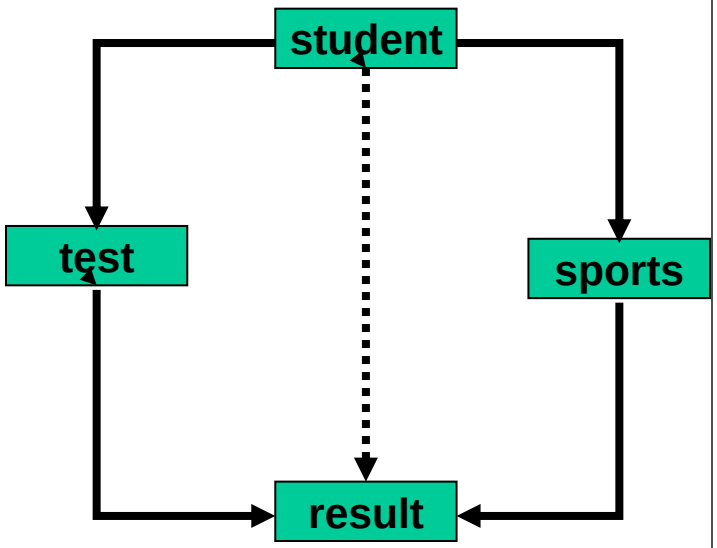
- All the public and protected members of student are inherited into result twice, first via test and again via sports.

- This means result class have duplicate set of members inherited from student.



Virtual Base Classes

```
class student
{
    ……
};
class test : virtual public student
{
    ……
};
class sports : public virtual student
{
    ……
};
class result : public test, public sports
{
    ……\
};
```



Abstract Classes

- An abstract class is one that is not used to create objects.
- An abstract class is designed only to act as a base class.
- It is a design concept in program development and provides a base upon which other classes may be built.

Constructors in Derived Classes

- If no base class constructor takes any arguments, the derived class need not have a constructor function.

- If any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

Constructors in Derived Classes

- When both the derived and base class contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

- In case of multiple inheritance, the base class constructors are executed in the order in which they appear in the declaration of the derived class.

Constructors in Derived Classes

- In a multilevel inheritance, the constructors will be executed in the order of inheritance.

- Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared.

Constructors in Derived Classes

- The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class.

- The base constructors are called and executed before executing the statements in the body of the derived constructor.

Constructors in Derived Classes

- The header line of *derived-constructor* function contains two parts separated by a colon (:).
  - The first part provides the declaration of the arguments that are passed to the derived constructor.
  - The second part lists the function calls to the base constructors.

Defining Derived Constructors

Derived-constructor(Arglist1, Arglist2, … ArglistN, ArglistD) :

base1(arglist1),

base2(arglist2),

…

baseN(arglistN)

{

}