Inheritance

By Shailee Shah
Assistant professor
President Institute of Computer Application

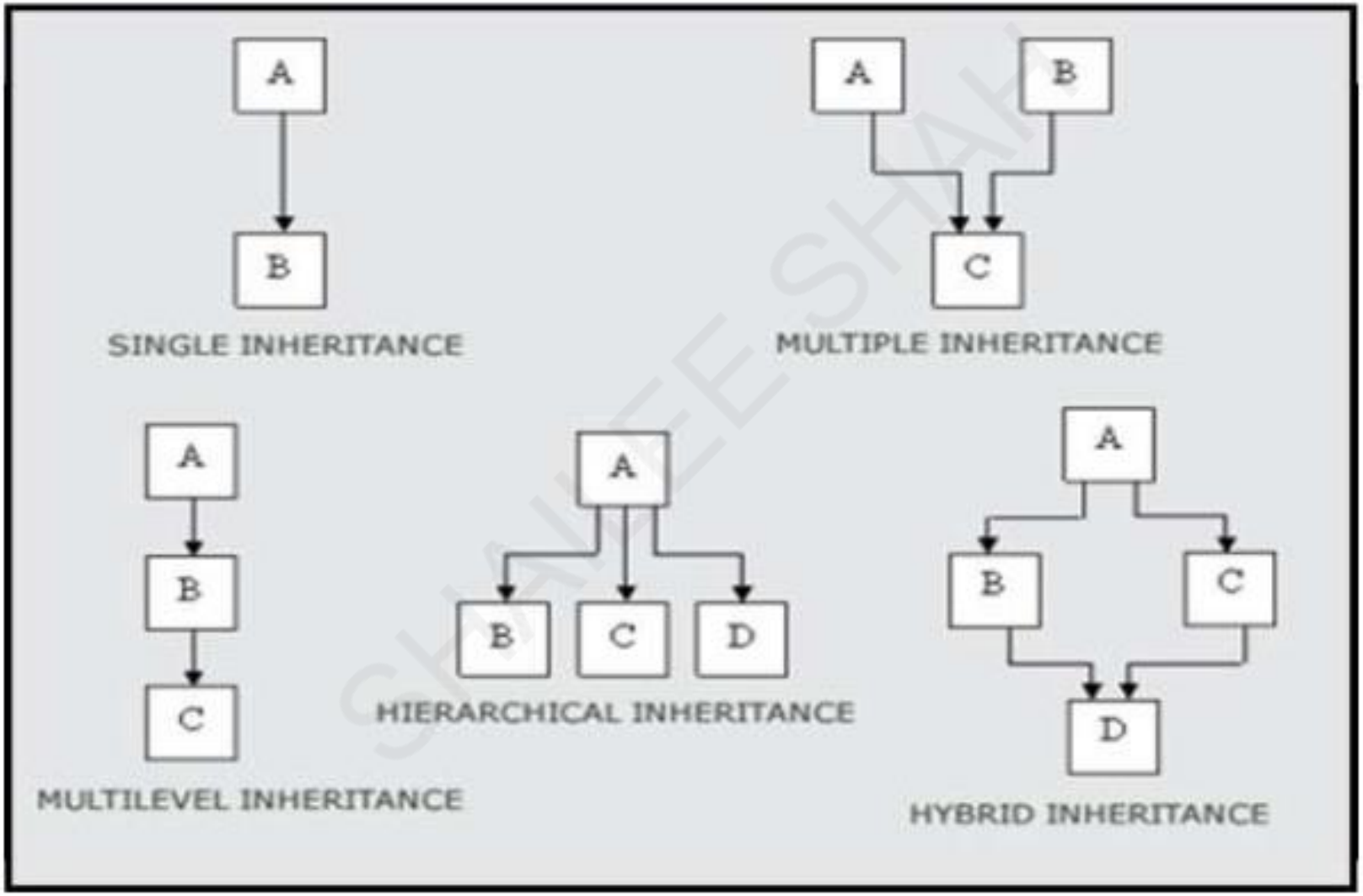
Inheritance

Reusability is an important feature of OOP and C++ strongly supports the concept of reusability.

Inheritance

- The C++ classes can be reused in several ways.
- Once a class has been written and tested, it can be adapted by other programmers to suit their requirements.
- This is basically done by creating new classes, reusing the properties of the existing classes.
- The mechanism of deriving a new class from an old class is called ***inheritance*** or ***derivation.***
- The old class is referred to as the ***base class*** and the new one is called the ***derived class*** or ***subclass.***

Types of inheritance



Defining Derived Classes

Syntax :

Class base class
{

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

Example :

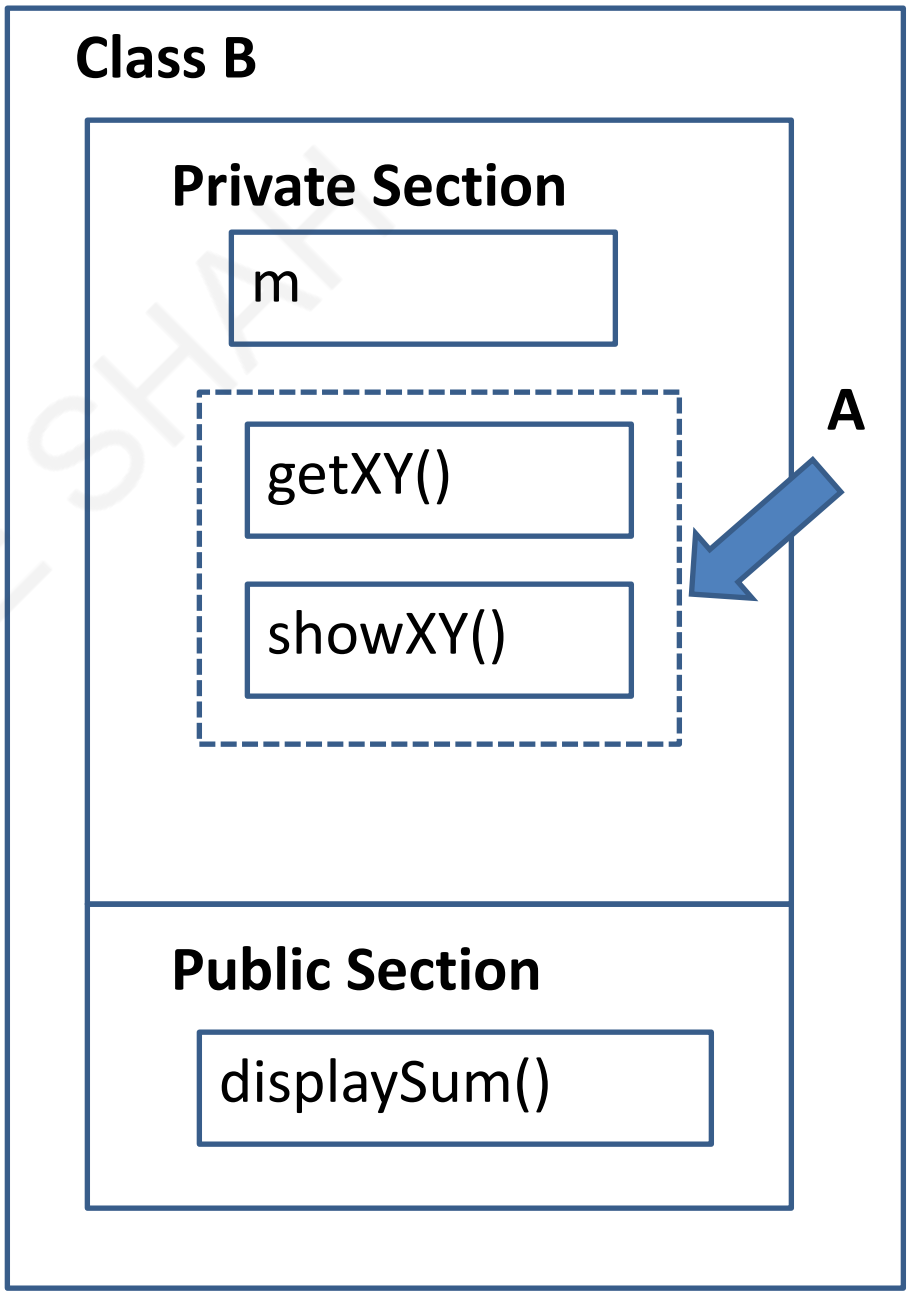
class ***B*** : ***public A***
{
// body of class B
}

Private Visibility-Mode

```
class A
{int x, y;
 public :
  void getXY();
  void showXY();};
```

```
class B
{
 int m;
 public :
    void displaySum();
};
```

```
class B : private A
{
member of derived
};
```



- **'public member'** of the base class become **'private member'** of the derived class
- **private member'** of the base class can not inherited by the derived class
- **'private member'** of base class can only be accessed by the **'member function'** of Base class.
- **'public member'** of base class can only be accessed by the **'member function'** of derived class.
- **'public member'** of a derived class can be accessed by its own object.
- **'public member'** of base class are not accessible to the **'objects'** of the derived class.

Public Visibility-Mode

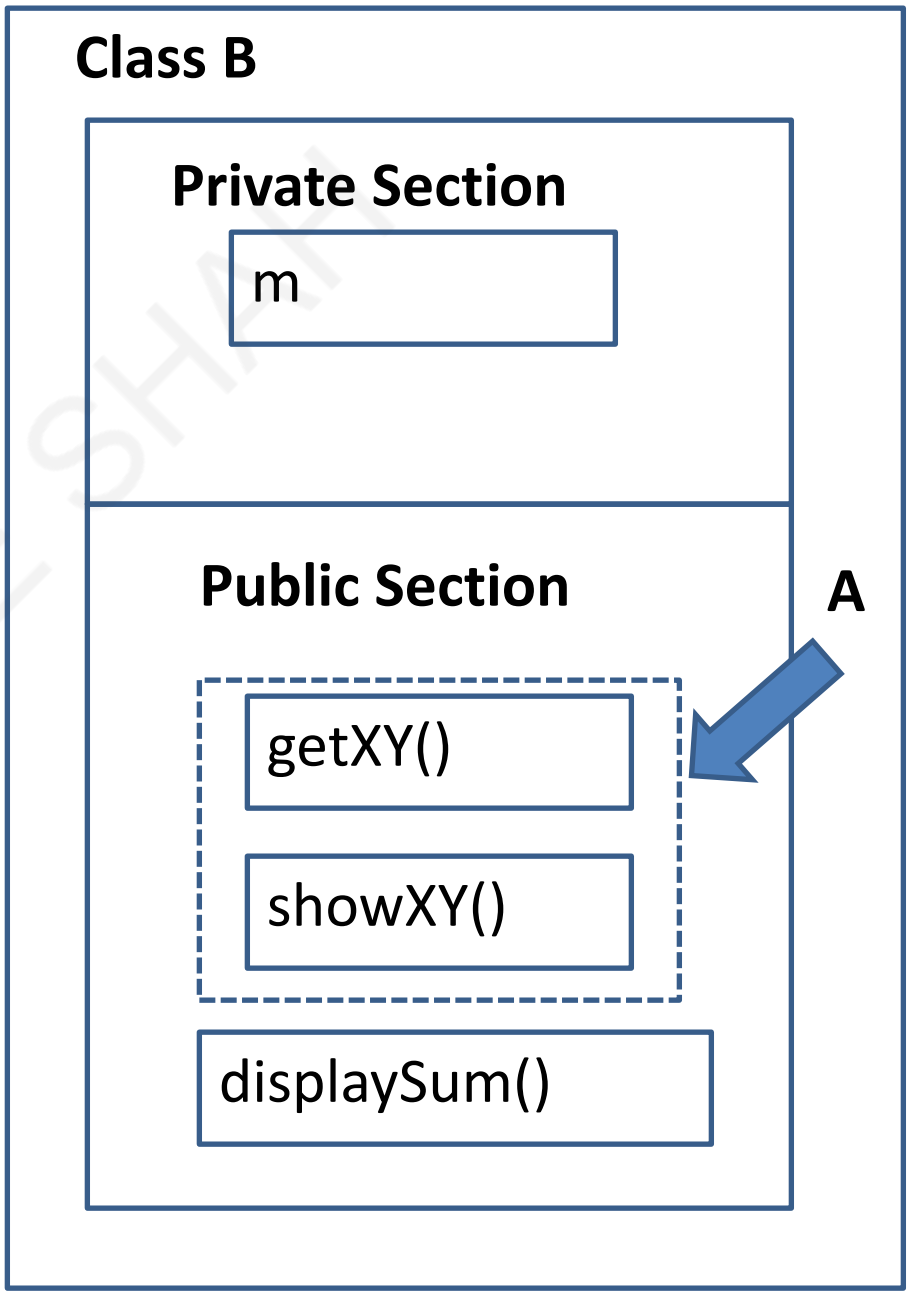
```
class A
{int x, y;
 public :
  void getXY();
  void showXY();};
```

```
class B
{int m;
 public :
    void displaySum(); };
```

```
class B : public A
{          member of derived          };
```

- **'public member'** of the base class become **'public member'** of the derived class
- **'public members'** of base class are accessible to the object of the derived class.

- **'Private members'** of based class never inherited into derived class in both **public** and **private** derivation



Types of Inheritance

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

Single Inheritance

A derived class with only one base class is called ***single inheritance.***

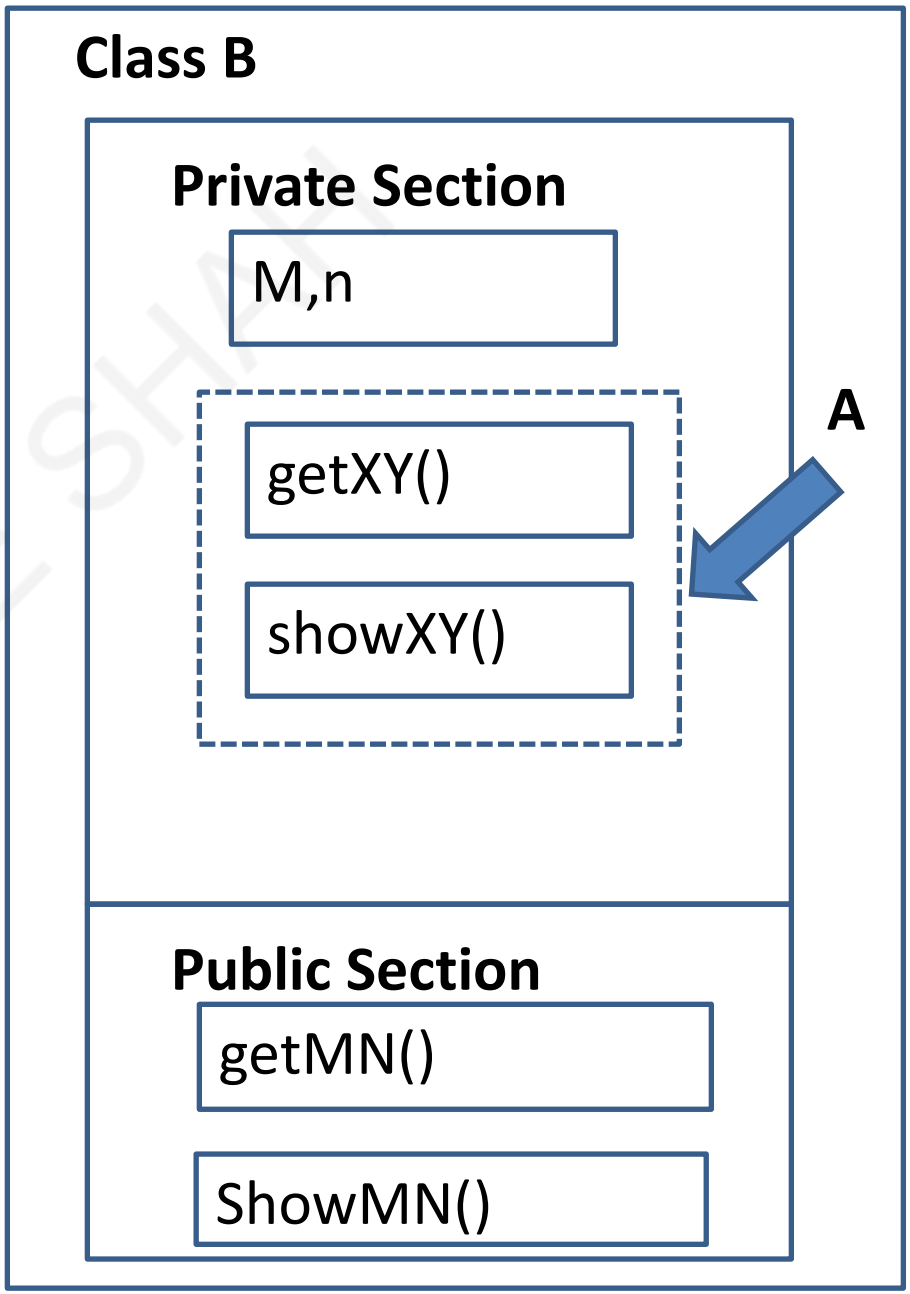
Single Inheritance

```
class A
{
        int x, y;
    public :
        void getXY(){
            x = 10;
            y = 20;
        }
        void showXY(){
            cout<<"\nX =
"<<x;
            cout<<"\nY =
"<<y;
        }
};
```

```
class B : private A
{
        int m, n;
    public :
        void getMN(){
            getXY();
            m = 30;
            n = 40;
        }
        void showMN(){
            showXY();
            cout<<"\nM = "<<m;
            cout<<"\nN = "<<n;
        }
};
```

```
void main(){
    B b1;
    //b1.getXY();
    b1.getMN();
    //b1.showXY();
    b1.showMN();
}
```

Output :
X = 10
Y = 20
M = 30
N = 40



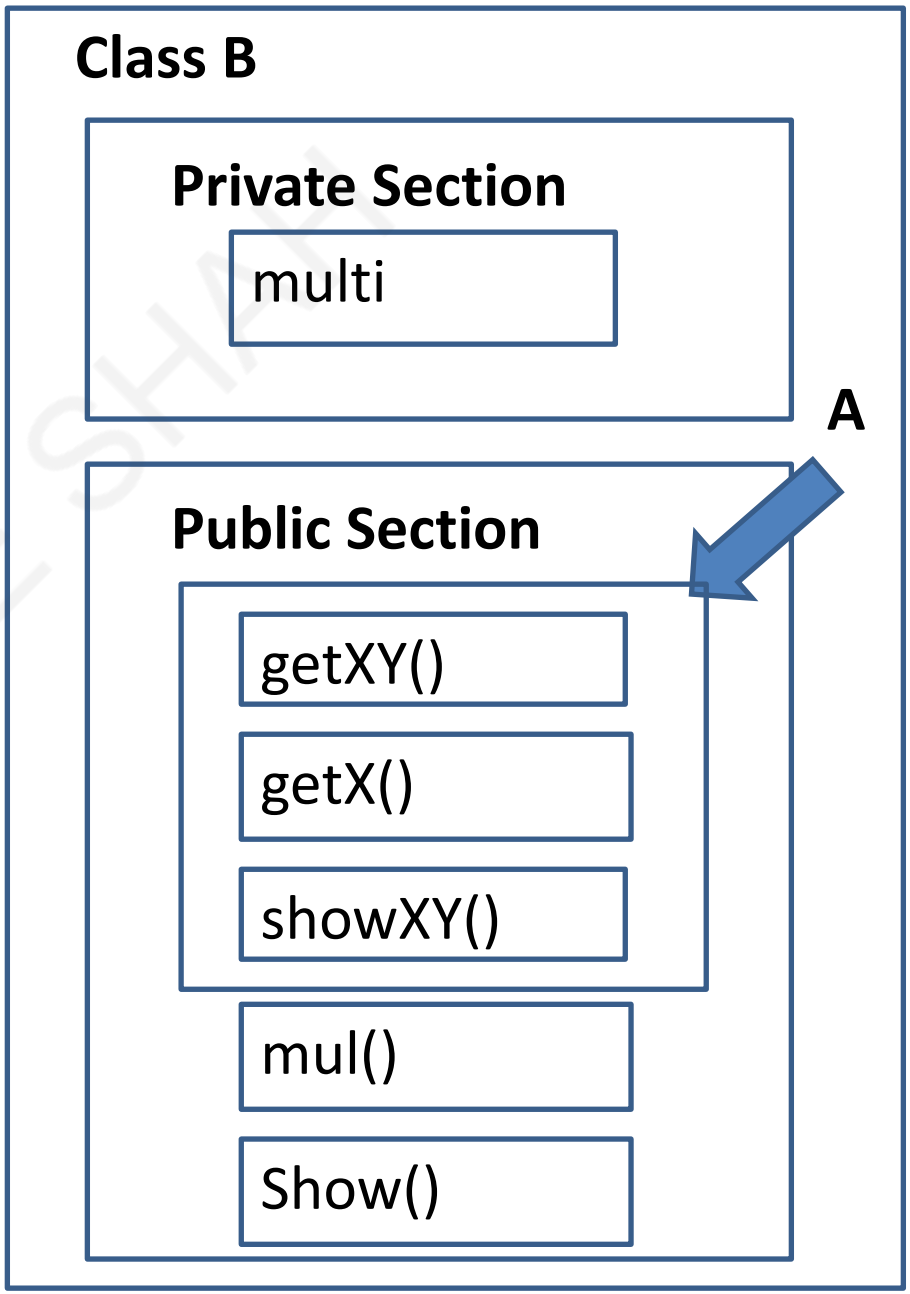
Single Inheritance

```
class A
{
        int x;
    public :
        int y;
        void getXY()
        {
            x = 5;y = 10;
        }
        int  getX(){
            return x;
        }

        void showX()
        {
            cout<<"\nX is  = "<<x;

        }
};
```

```
class B : public  A
{
        int multi;
    public :
        void mul()
        {
            multi=y*getX();
        }
        void show()
        {
            showX();
            cout<<"\Y = "<<y;
            cout<<"\n
            Multiplication =
            "<<multi;
        }
};
```

```
void main(){
    B b1;
    b1.getXY();
    b1.mul();
    b1.showX();
    b1.show();
    //b1.x=20
    b1.y=20;
    b1.mul();
    b1.show();
}
```

```
Output :
X is = 5
X is = 5
Y = 10
Multiplication=50
X is = 5
y = 20
Multiplication=100
```



Protected Visibility-Mode

- C++ provides a third visibility modifier, Protected, Which serve a limited purpose in inheritance.
- A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.
- It can not be accessed by the functions outsides these two classes.

Protected Visibility-Mode

- When **protected member** is inherited in **public mode,**

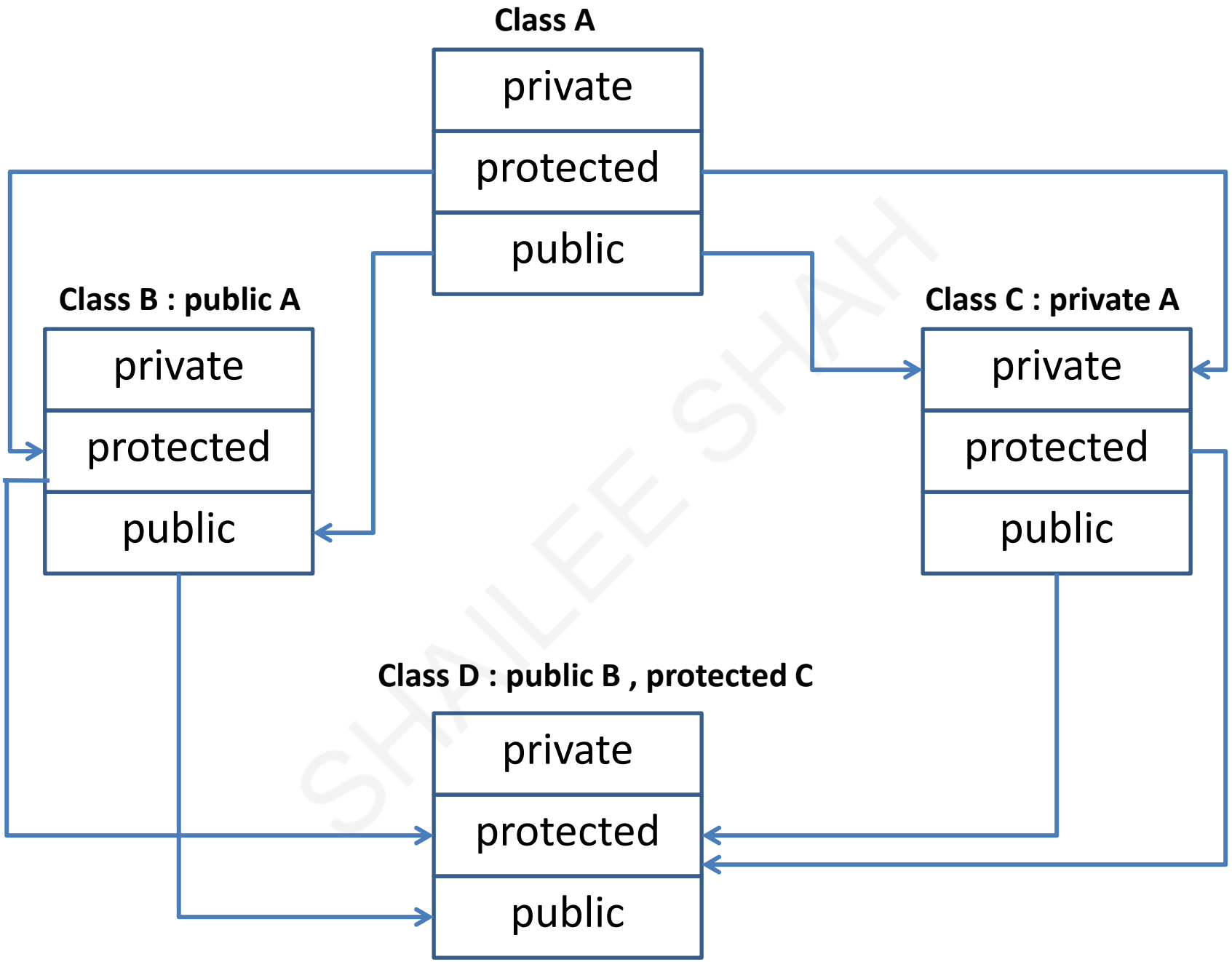
  'protected member' of the base class become **'protected member'** of the derived class and therefore is only accessible by the member function of the derived class.

  It is also ready for further inheritance.

- When **protected member** is inherited in **private mode,**

  'protected member' of the base class become **'private member'** of the derived class and therefore is only accessible by the member function of the derived class.

  it is not available for further inheritance.

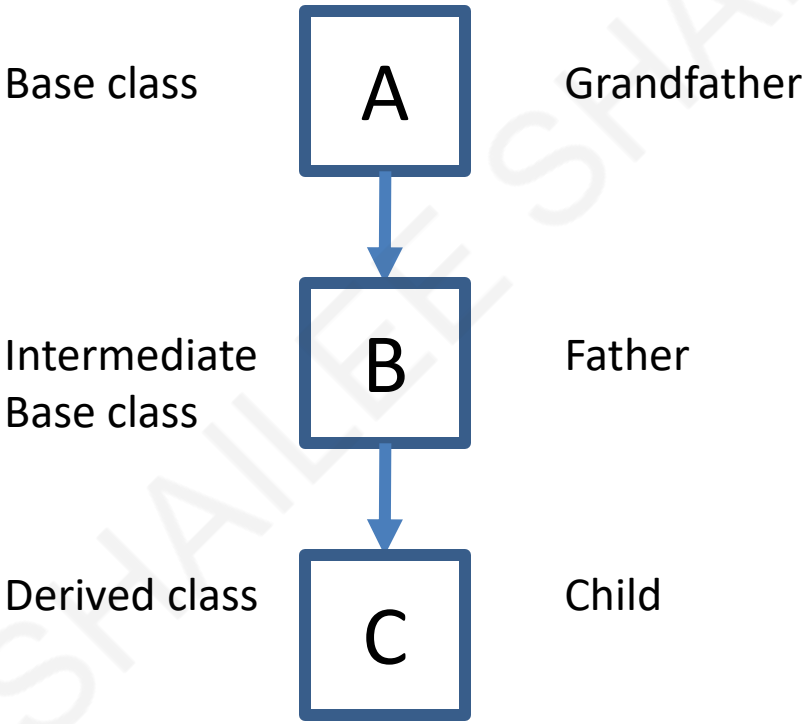

Class A
private
protected
public
Class B : public A
private
protected
public
Class C : private A
private
protected
public
Class D : public B , protected C
private
protected
public
SHAILEE SHAH

Visibility of inherited member

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

- Private members of based class never inherited into derived class in any derivation

Multilevel Inheritance



Mechanism of deriving a class from another derived class is known as **multilevel inheritance.**

```
class student
{
   protected :
      Int roll_no;
   public :
      void get_num(int a)
      {
         roll_no=a;
      }
      void put_num()
      {
         cout<<"\n Roll no is
         = "<<roll_no;
      }
};
```

```
class test : public student
{
   protected :
      float sub1;
      float sub2;
   public :
      void get_marks(float x,float y)
      {
         sub1=x;
         Sub2=y;
      }
      void put_marks()
      {
         cout<<"\n marks in sub1 is
         = "<<sub1;
         cout<<"\n marks in sub2 is
         = "<<sub2;
      }
};
```

```
class result : public test{
   protected :
      float total;
   public :
      void display()
      {
         total=sub1+sub2;
         put_num();
         put_marks();
         cout<<"\n total is
         = "<<total;
      }
};
```

```
void main(){
   result r;
   r.get_num(101);
   r.get_marks(50,70);
   r.display();
}
```

Multilevel Inheritance

```
class A
{
      int x, y;
   protected :
      void getA(){
         x = 10;
         y = 20;
      }
   public :
      void showA(){
         cout<<"\nX = "<<x;
         cout<<"\nY = "<<y;
      }
};
```

```
class B : public A
{
      int m, n;
   public :
      void getB(){
         getA();
         m = 30;
         n = 40;
      }
      void showB(){
         cout<<"\nM = "<<m;
         cout<<"\nN = "<<n;
      }
};
```

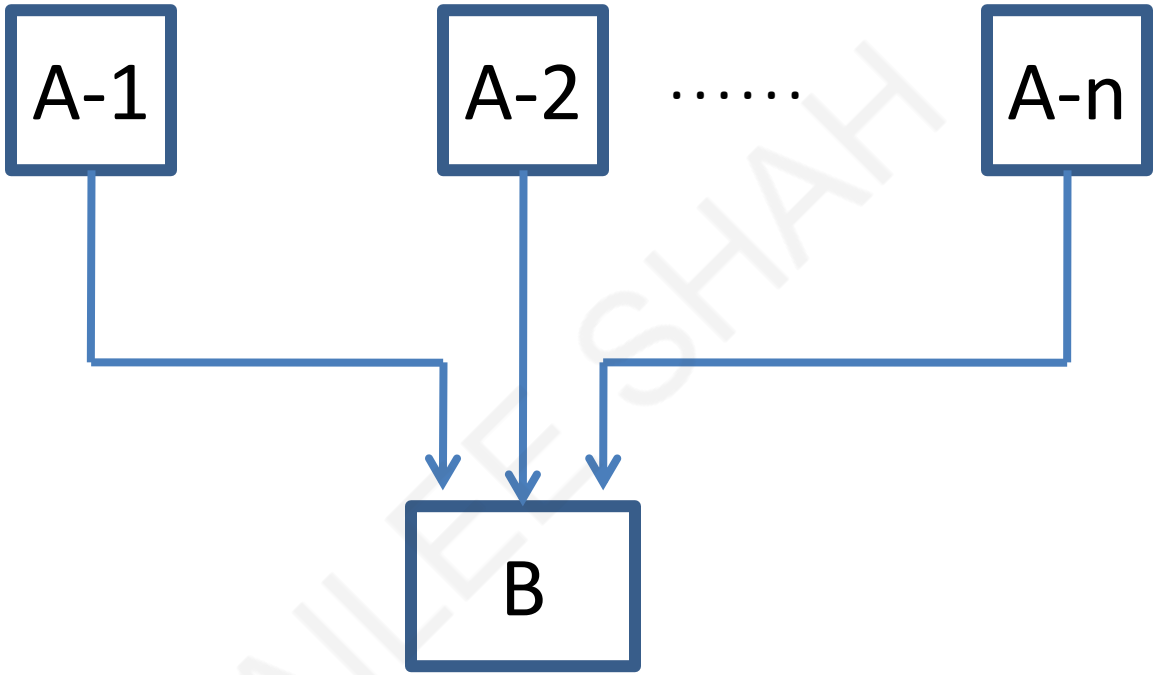
```
class C : public B{
      int x1, y1;
   public :
      void getC(){
         x1 = 50;
         y1 = 60;
      }
      void showC(){
         getA();
         cout<<"\nX1 = "<<x1;
         cout<<"\nY1 = "<<y1;
      }
};
```

Multilevel Inheritance

```
void main(){
   C c1;
   c1.getB();
   c1.getC();
   c1.showA();
   c1.showB();
   c1.showC();
}
```

```
Output :

X is = 10
y is = 20
m is = 30
N is = 40
X1 is = 50
Y1 is = 60
```

Multiple Inheritance



- A class can inherit the attributes of two or more classes. This is known as **multiple inheritance.**
- This allows us to combine the features of several existing classes as a starting point for defining new classes.
- It is like a child inheriting the physical features of one parent and the intelligence of another.

Syntax

```
class <derived class> : [visibility] <based class1>, [visibility] <based class2>, . . . . .
{
        . . . . . .
        . . . . . .   (Body of B)
        . . . . . .
}
```

```
class A{
      int a1, a2;
   public:
      void setA(int x, int y){
         a1 = x; a2 = y;
      }
      void putA(){
         cout<<"\nA1 = "<<a1;
         cout<<"\nA2 = "<<a2;
      }
};
```

```
class B{
      int b1, b2;
   public:
      void setB(int x, int y){
         b1 = x; b2 = y;
      }
      void putB(){
         cout<<"\nB1 = "<<b1;
         cout<<"\nB2 = "<<b2;
      }
};
```

```
class C : public A, private B{
      int c1, c2;
   public:
      void setC(int x, int y){
         setB(30, 40);
         c1 = x; c2 = y;
      }
      void putC(){
         putB();
         cout<<"\nC1 = "<<c1;
         cout<<"\nC2 = "<<c2;
      }
};
```

Multiple Inheritance

```
void main(){
   C c1;
   c1.setA(10, 20);
   c1.putA();
   c1.setC(50, 60);
   c1.putC();
}
```

```
Output :
A1 is = 10
A2 is = 20

B1 is = 30
B2 is = 40

C1 is = 50
C2 is = 60
```

```
class A{
      int a1, a2;
   public:
      void setA(int x, int y){
         a1 = x; a2 = y;
      }
      void show(){
         cout<<"\nA1 = "<<a1;
         cout<<"\nA2 = "<<a2;
      }
};
```

```
class B{
      int b1, b2;
   public:
      void setB(int x, int y){
         b1 = x; b2 = y;
      }
      void show(){
         cout<<"\nB1 = "<<b1;
         cout<<"\nB2 = "<<b2;
      }
};
```

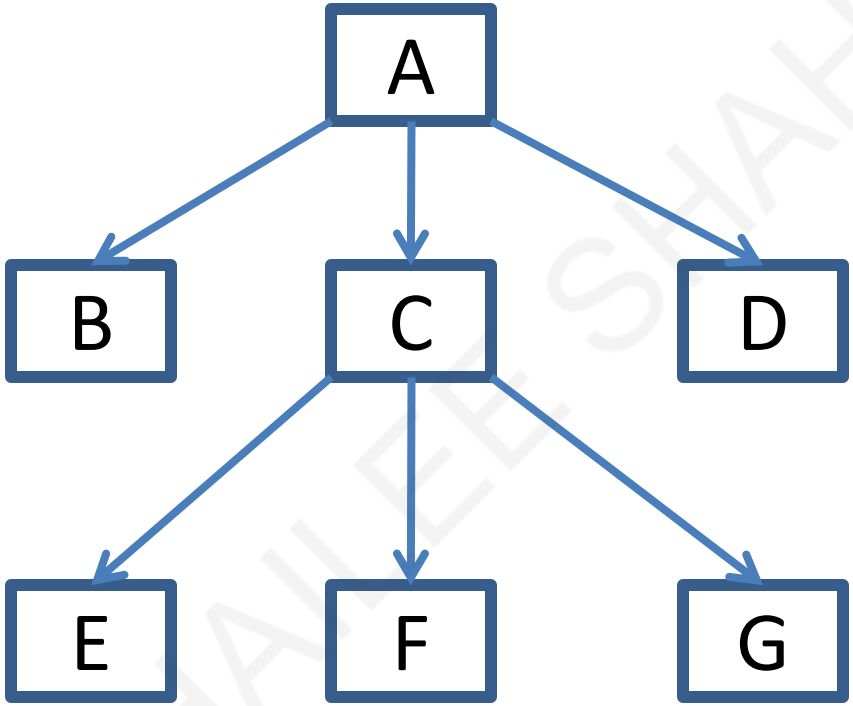
```
class C : public A, private B{
      int c1, c2;
   public:
      void setC(int x, int y){
         setB(30, 40);
         c1 = x; c2 = y;
      }
      void show(){
         A :: show();
         B :: show();
         cout<<"\nC1 = "<<c1;
         cout<<"\nC2 = "<<c2;
      }
};
```

Inheritance with function Overriding

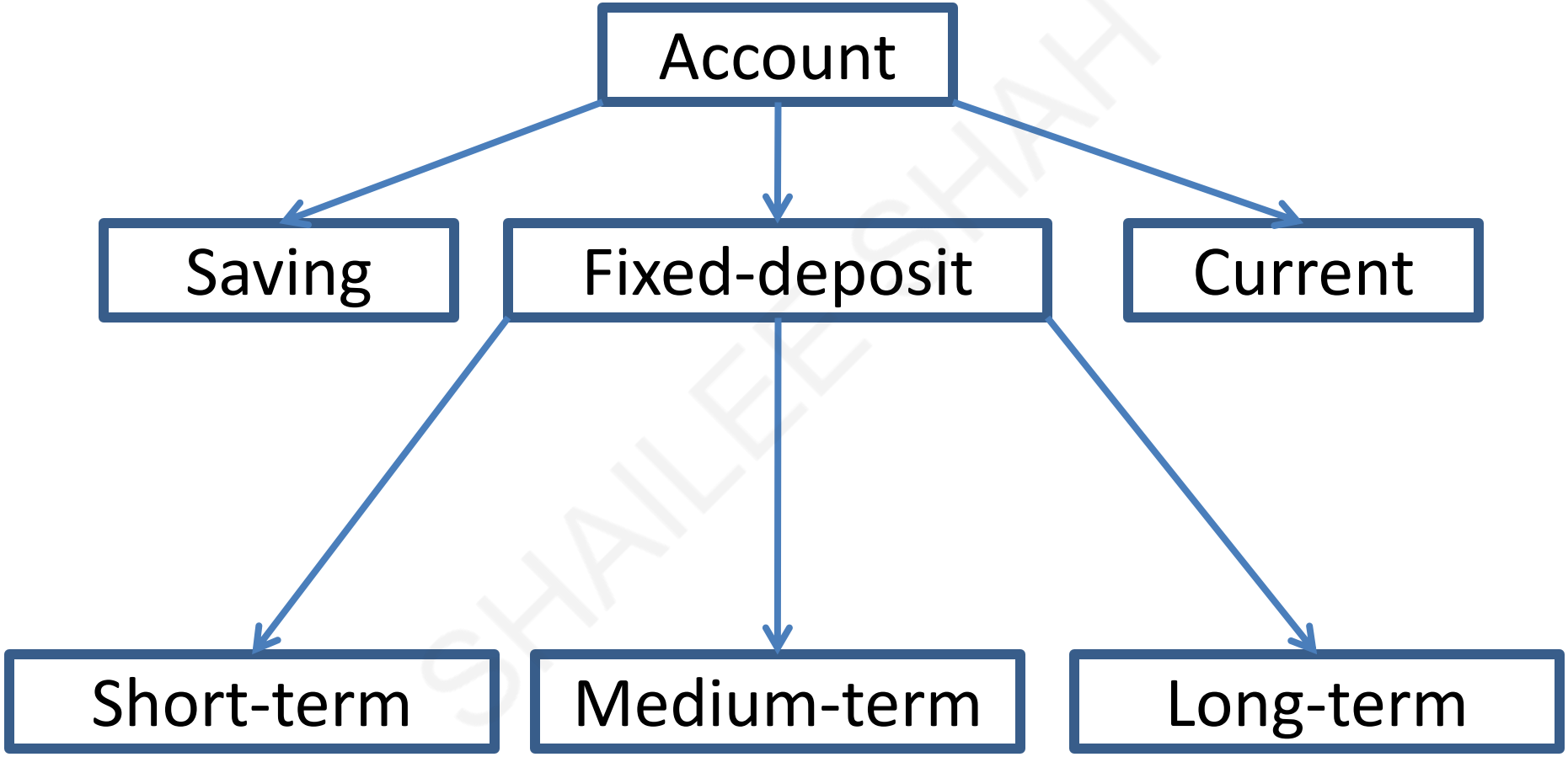
```
Output :
A1 is = 10
A2 is = 20

B1 is = 30
B2 is = 40

C1 is = 50
C2 is = 60
```

```
void main(){
   C c1;
   c1.setA(10, 20);
   c1.setC(50, 60);
   c1.show();
}
```

Hierarchical Inheritance



Many Programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.



Hierarchical Inheritance

- In C++, such problems can be easily converted into class hierarchies.
- The base class will include all the features that are common to the subclasses.
- A subclass can be constructed by inheriting the properties of the base class.
- A subclass can serve as a base class for the lower level classes and so on.

```
class Number
{
   private:
      int num;
   public:
      void getNumber(void)
      {
         cout << "Enter an integer number: ";
         cin  >> num;
      }
      //to return num
      int returnNumber(void)
      { return num; }
};

//Base Class 1, to calculate square of a number
class Square:public Number
{
   public:
   int getSquare(void)
   {
      int num,sqr;
      num=returnNumber(); //get number from class Number
      sqr=num*num;
      return sqr;
   }
};

//Base Class 2, to calculate cube of a number
class Cube:public Number
{
   private:

   public:
   int getCube(void)
   {
      int num,cube;
      num=returnNumber(); //get number from class Number
      cube=num*num*num;
      return cube;
   }
};
int main()
{
      Square objS;
      Cube objC;
      int sqr,cube;

      objS.getNumber();
      sqr =objS.getSquare();
      cout << "Square of "<< objS.returnNumber() << " is: " << sqr  <<
        endl;
      objC.getNumber();
      cube=objC.getCube();
      cout << "Cube   of "<< objS.returnNumber() << " is: " << cube <<
        endl;
          return 0;
}
```

Output :

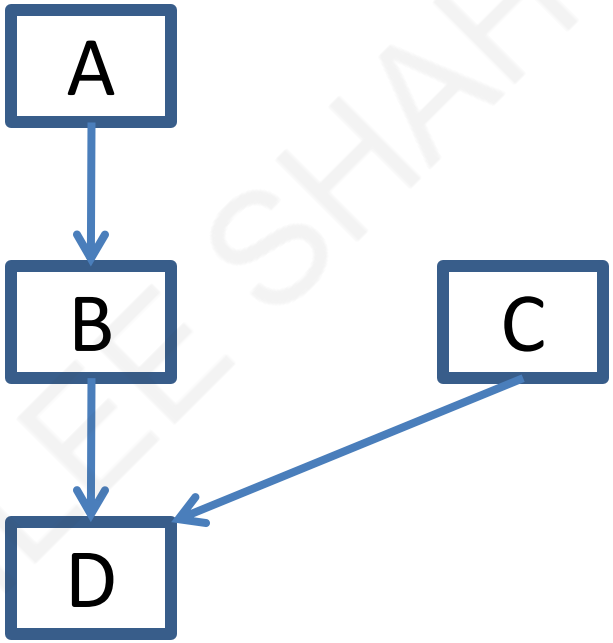
Enter an integer number: 10

Square of 10 is: 100

Enter an integer number: 20

Cube of 20 is: 8000

Hybrid Inheritance



The combination of two or more inheritance is known as **Hybrid Inheritance.**

Hybrid Inheritance

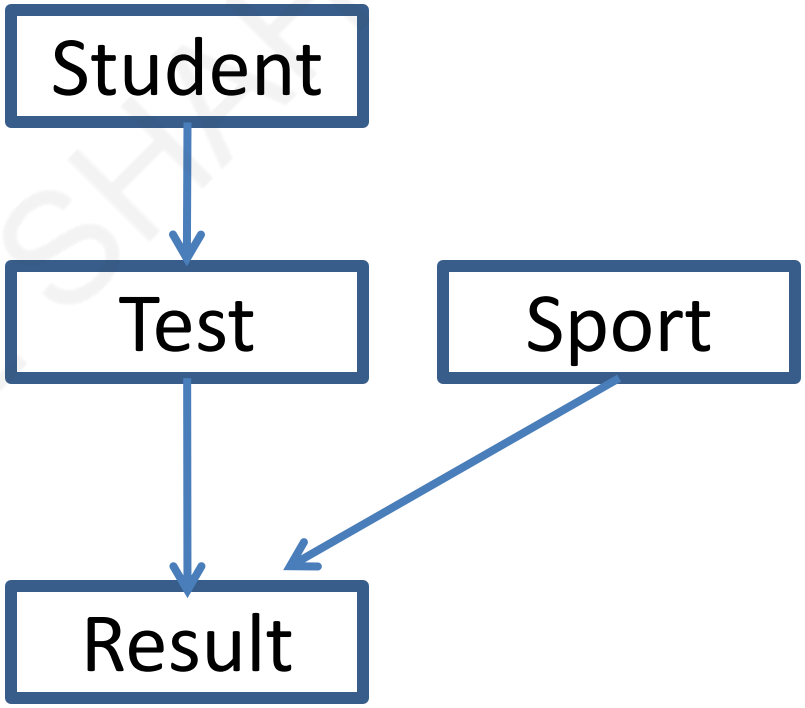
There could be situation we need to apply two or more types of inheritance to design a program.

For example,

Consider the case of processing the student results.

Assume that we have to give weightage for sports before finalizing the results.

The weightage for sport is stored in a separate class called sports.



```
class student
{
   protected :
      Int roll_no;
   public :
      void get_num(int a)
      {
         roll_no=a;
      }
      void put_num()
      {
         cout<<"\n Roll no is
         = "<<roll_no;
      }
};
```

```
class test : public student
{
   protected :
      float sub1;
      float sub2;
   public :
      void get_marks(float x,float y)
      {
         sub1=x;
         Sub2=y;
      }
      void put_marks()
      {
         cout<<"\n marks in sub1 is
         = "<<sub1;
         cout<<"\n marks in sub2 is
         = "<<sub2;
      }
};
```

```
Class sport : public student
{
   protected :
      float sport_sub;
   public :
      void
get_sportmarks(float x)
      {
               sport_sub=x;
      }
      void put_sportmarks()
      {
         cout<<"\n marks in
sports subject is
         = "<< sport_sub;
      }
};
```

Hybrid Inheritance

```
class result : public test, public sports
{
   protected :
      float total;
   public :
      void display()
      {
         total=sub1+sub2+sport_sub;
         put_num();
         put_marks();
         put_sportmarks();
         cout<<"\n total is   = "<<total;
      }
};
```

Hybrid Inheritance

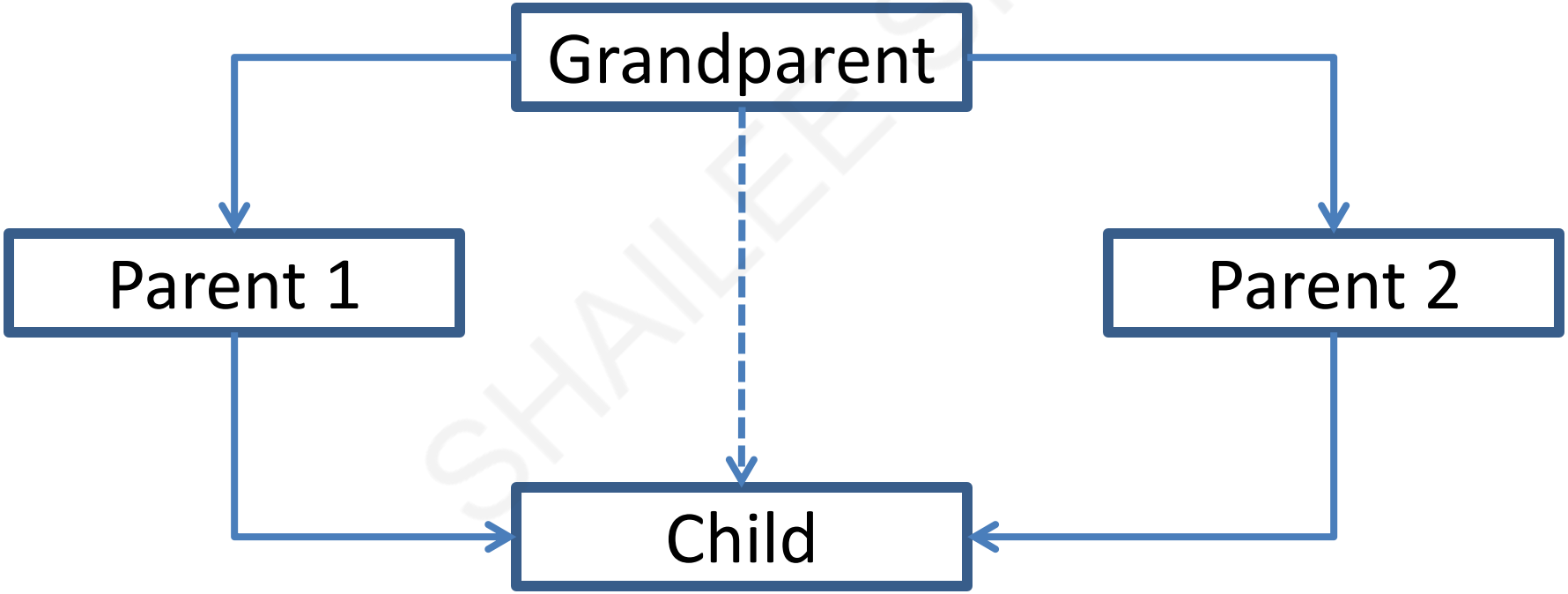
```
void main()
{
   result r;
   r.get_num(101);
   r.get_marks(50,70);
   r.get_sportmarks(80);
   r.display();
}
```

Output :

Roll no is = 101
marks in sub1 is = 50
marks in sub2 is = 70
marks in sport subject is = 80
Total is = 200

Virtual Base Class

- Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved as follow :



Virtual Base Class

- The ‘child’ has two direct base classes ‘parent1’ and ‘parent2’ which themselves have a common base class ‘grandparent’.
- The ‘child’ inherits the traits of ‘grandparent’ via two separate paths.
- It can also inherit directly from ‘grandparent’. So ‘grandparent’ is sometimes referred to as indirect base class.
- Inheritance by the child as this way might pose some problems.
- All the public and protected member of ‘grandparent’ are inherited into ‘child’ twice,

  first via ‘parent1’ and again via ‘parent2’.
- This means, ‘child’ would have duplicate sets of the members inherited from ‘grandparent’.
- This introduces ambiguity and should be avoided.

Virtual Base Class

- The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class while declaring the direct or intermediate base classes.

```
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
};
```

Note : The keywords ***virtual*** and ***public*** may be used in either order.

Virtual Base Class

- When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

```
class student
{
   protected :
      Int roll_no;
   public :
      void get_num(int a)
      {
         roll_no=a;
      }
      void put_num()
      {
         cout<<"\n Roll no is
         = "<<roll_no;
      }
};
```

```
class test : virtual public student
{
   protected :
      float sub1;
      float sub2;
   public :
      void get_marks(float x,float y)
      {
         sub1=x;
         Sub2=y;
      }
      void put_marks()
      {
         cout<<"\n marks in sub1 is
         = "<<sub1;
         cout<<"\n marks in sub2 is
         = "<<sub2;
      }
};
```

```
class sport : public virtual
student
{
   protected :
      float sport_sub;
   public :
      void
get_sportmarks(float x)
      {
               sport_sub=x;
      }
      void put_sportmarks()
      {
         cout<<"\n marks in
sports subject is
         = "<< sport_sub;
      }
};
```

Virtual base class

```
class result : public test, public sports
{
   protected :
      float total;
   public :
      void display()
      {
         total=sub1+sub2+sport_sub;
         put_num();
         put_marks();
         put_sportmarks();
         cout<<"\n total is   = "<<total;
      }
};
```

Hybrid Inheritance

```
void main()
{
   result r;
   r.get_num(101);
   r.get_marks(50,70);
   r.get_sportmarks(80);
   r.display();
}
```

Output :

Roll no is = 101
marks in sub1 is = 50
marks in sub2 is = 70
marks in sport subject is = 80
Total is = 200

```
class A{
      int a1, a2;
   public:
      void setA(int x, int y){
         a1 = x; a2 = y;
      }
      void putA(){
         cout<<"\nA1 = "<<a1;
         cout<<"\nA2 = "<<a2;
      }
};
```

```
class B : virtual public A{
      int b1, b2;
   public:
      void setB(int x, int y){
         b1 = x; b2 = y;
      }
      void putB(){
         cout<<"\nB1 = "<<b1;
         cout<<"\nB2 = "<<b2;
      }
};
```

```
class C : public virtual A{
      int c1, c2;
   public:
      void setC(int x, int y){
         c1 = x; c2 = y;
      }
      void putC(){
         cout<<"\nC1 = "<<c1;
         cout<<"\nC2 = "<<c2;
      }
};
```

Virtual Base Class

```
class D : public B, public C{

};
```

```
void main(){
   clrscr();
   D d1;
   d1.setA(10, 20);
   d1.putA();
   getch();
}
```

Abstract Classes

- An abstract class in one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes.)
- It is a design concept in program development and provides a base upon which other classes may be built.
- The class which contains at least one pure virtual function that class called as abstract class only.

```
class A{
  public:
    virtual void display()=0;
};
```

```
class B : public A{
    int b1, b2;
  public:
    void setB(int x, int y){
      b1 = x; b2 = y;
    }
    void display(){
      cout<<"\nB1 = "<<b1;
      cout<<"\nB2 = "<<b2;
    }
};
```

```
class C : public A{
    int c1, c2;
  public:
    void setC(int x, int y){
      c1 = x; c2 = y;
    }
    void display(){
      cout<<"\nC1 = "<<c1;
      cout<<"\nC2 = "<<c2;
    }
};
```

Abstract Class

```
void main(){
  clrscr();
  B b1;
  b1.setB(30, 40);
  b1.display();
  C c1;
  c1.setC(50, 60);
  c1.display();
  getch();
}
```

Abstract Class

```
class vehicle{
   public:
      virtual void sp_parts()=0;
};
```

```
class  car: public vehicle{
         void sp_parts()
        {
              cout<<“CAR
CLASS”;
        }
};
```

```
class  activa : public vehicle{
      void sp_parts()
        {
              cout<<“ACTIVA
CLASS”;
        }
};
```

Member Classes : Nesting of Classes

- C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects.

```
Class A {. . . . . .};
Class B {. . . . . .};
Class C {
            A a1;
            B  b1;
            . . . . . .
};
```

Polymorphism

Polymorphism

- Polymorphism is one of the crucial features of OOP.
- It simply means ‘one name, multiple forms.’
- We have already seen how the concept of polymorphism is implemented using the overloaded functions and operators.
- The overloaded member functions are ‘selected’ for invoking by matching arguments, both type and number.
- This information is know to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself.
- This is called ***early binding*** or ***static binding*** or ***static linking***. Also known as ***compile time polymorphism***.
- Early binding simply means that an object is bound to its function call at compile time.

Polymorphism

- Consider following example,

```
Class A{
      int x;
   public:
      void show(){ . . . . . .}
};
Class B{
      int y;
   public:
      void show(){ . . . . . .}
};
```

- Here, prototype of ***show()*** is the same in both the classes, the function is not overloaded and therefore static binding does not apply.

Polymorphism

- It would be nice if the appropriate member function could be selected while program is running.
- This is known as run time polymorphism.
- At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked.
- Since the function is linked with a particular class much later after the selection of the appropriate function is done dynamically at run time.
- Dynamic binding is one of the powerful features of C++.
- This requires the use of pointers to objects.

Pointers

- Pointers is one of the key aspect of C++ language similar to that of C.
- We know that a pointer is a derived data type that refers to another data variable by storing the variable's member address rather than data.
- A pointer variable defines where to get the values of a specific data variable instead of defining actual data.

Declaring and Initializing Pointers

- We can declare a pointer variable similar to other variable in C++.
- Like C, the declaration is based on the data type of the variable it points to.
- Syntax :

  data-type *pointer-variable;
- Example :

  int *ptr;
- We can initialize a pointer variable as follows :

  int *ptr, a;

  ptr = &a;

Manipulation of Pointers

```
void main(){
        int a = 10;
        int *ptr;
        ptr = &a;
        cout<<“A is = “<<a;
        *ptr = *ptr + a; //*ptr and a both have value of a
        cout<<“\nNew A is = “<<*ptr;
}

//*ptr and a both have value of a
// ptr and &a both have address of a
```

Pointer Expressions and pointer Arithmetic

- There are a substantial number of arithmetic operations that can be perform with pointer.
- C++ allows pointer to perform the following arithmetic operations
  - A pointer can be incremented (++) (or) decremented (--)
  - Any integer can be added to or subtract from a pointer
  - One pointer can be subtracted from another.
- Example,

  int a[6]; int *ptr; ptr = &a;
- Here the pointer variable ***ptr*** refers to the base address of the variable **a**. We can increment of decrement the pointer as follows:

  ptr++ (or) ++ptr ptr-- (or) --ptr
- This statements moves the pointer to the next memory address. Also if two pointer variable point to the same array they can be subtract from each other.

Pointer Expressions and pointer Arithmetic

```
void main(){
   clrscr();
   int num[] = {22,33,44,55,66};
   int *ptr, i;
   cout<<"The array is = \n";
   for(i=0;i<5; i++)    cout<<num[i]<<"\n";

   ptr = num;
   cout<<"\nValue of ptr = "<<*ptr;       //22
   ptr++;//ptr[0]---ptr[1]
   cout<<"\nValue of ptr = "<<*ptr;       //33
   ptr--;//ptr[1]---ptr[0]
   cout<<"\nValue of ptr = "<<*ptr;       //22
   ptr = ptr+2;//ptr[0]---ptr[2]
   cout<<"\nValue of ptr = "<<*ptr;       //44
   ptr = ptr-2;///ptr[2]---ptr[0]
   cout<<"\nValue of ptr = "<<*ptr;       //22
   getch();
}
```

Pointers to Objects

- We have seen a pointer can point to an object created by a class.

  item x;

  item *it_ptr;

- Object pointers are useful in creating objects at run time. we can also use an object pointer to access the public members of an object.

```
class item{
      int code;
      float price;
   public:
      void getdata(int a, float b){
         code = a;  price = b;
      }
      void show(){
         cout<<“Code : “<<code<<“\n”;
         cout<<“Price : “<<price<<“\n”;
      }
}
```

Pointers to Objects

- Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows :

  item x;

  item *ptr = &x;

  The pointer ptr is initialized with the address of x.
- We can refer to the member functions of item in two ways, one by using the dot operator and the object, and another by using the arrow operator and the object pointer.
- The statements

  x.getdata(10,20,30); x.show();

  are equivalent to,

  ptr->getdata(10,20,30); ptr->show();
- Since ***ptr** is an alias of **x**, we can also use the following method :

  (*ptr).show();

Note : The parentheses are necessary because the **dot operator** has higher precedence than the **indirection operator *.**

Pointers to Objects

- We can also create objects using pointer and new operator as follow :

  item *ptr = new item;

  This statement allocates enough memory for the data member in the object structure and assigns the address of the memory space to **ptr**.

  Then **ptr** can be used to refer to the members as shown below :

  ptr->show();

- If a class has a constructor with arguments and does not include an empty constructor, then we must supply the argument when the object is created.

Pointers to Objects

- We can also create an array of objects using pointer as follow :

  item *ptr = new item[10]; // array of 10 objects
- It will create memory space for an array of 10 objects of **item**.
- In such case, if the class contains constructors, it must also contain an empty constructor.

```
class A{
      int x, y;
   public:
      void getA(int a, float b){
         x = a;  y = b;
      }
      void showA(){
         cout<<"\nA : "<<x<<"\n";
         cout<<"\nB : "<<y<<"\n";
      }
};
```

```
void main(){
   clrscr();
   A *a1 = new A[2];
   A *a2 = a1;
   int p,q;
   for(int i=0; i<2; i++){
      cout<<"\nEnter A"<<i+1;
      cin>>p>>q;
      a1->getA(p, q);
      a1++;
   }
   for(i=0; i<2; i++){
      cout<<"\nThe A"<<i+1;
      a2->showA();
      a2++;
   }
   getch();
}
```

Pointer to Objects

this Pointer

- C++ uses a unique keyword called **this** to represent an object that invokes a member function.
- **this** is a pointer that points to the object for which this function was called.
- Exa, : The function call **A.max()** will set the pointer **this** to the address of the object A.
- This unique pointer is automatically passed to a member function when it is called.
- The pointer **this** acts as an implicit argument to all the member function.
- The private variable '**a**' can be used directly inside a member function, like

  a = 123;
- We can also use the following statement to do the same job :

  this->a = 123;

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

class A{
   int x, y;
   public :
      void setA(int x, int y){
         this->x = x;
         this->y = y;
      }
      void putA(){
         cout<<"\nX = "<<x;
         cout<<"\nY = "<<y;
      }
};
```

```
void main(){
   clrscr();
   A a1;
   a1.setA(10, 20);
   a1.putA();
   getch();
}
```


Use of this Pointer

this Pointer

Note : C++ permits the use of shorthand form a = 123, we have not been using the pointer this explicitly.

- However, we have been implicitly using the pointer **this** when overloading the operators using member.
- When a binary operator is overloaded using member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this**.
- One important application of the pointer **this** is to return it points to.

```
return *this;
```

- Inside a member function will return the object that invoked the function.

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

class A{
   int x;
   public:
      A(int p){
         x = p;
      }
      A & max(A & a){
         if(a.x >= x)
               return a;
         else
               return *this;
      }
      void show(){
         cout<<"x = "<<x;
      }
};
```

```
void main(){
   clrscr();
   A a1(20), a2(50);
   A a3 = a1.max(a2);
   a3.show();
   getch();
}
```


Use of Pointer

Pointers to Derived Classes

- We can use pointers not only to the base objects but also to the objects of derived classes.
- Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes.
- Exa, :

  If **B** is base class and **D** is derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**.

```
B *bptr;        //  pointer to class B type variable
B b;            //  base object
D d;            //  derived object
bptr = &b;      //  bptr points to object b
```

Pointers to Derived Classes

- We can make **bptr** to point to the object **d** as follows :

  bptr = &d; // **bptr** points to object **d**
- This is perfectly valid with C++ because **d** is an object derived from the class **B**.
- We can access only those member which are inherited from B and not the member that originally belong to **D**.
- In case a member of **D** has the same name as one of the members of **B**, then any reference to that member by **bptr** will always access the base class member.
- Although C++ permits a base pointer to point to any object derived from base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

```
class A{
   int x, y;
   void getA(int a, int b){
      x = a;
      y = b;
   }
   public:
      void showA(){
         getA(10, 20);
         cout<<"\nx = "<<x;
         cout<<"\ny = "<<y;
      }
};
```

```
class B : public A{
   int a, b;
   public:
      void getB(int x, int y){
         a = x;
         b = y;
      }
      void showB(){
         getB(30, 40);
         cout<<"a = "<<a;
         cout<<"b = "<<b;
      }
};
```

```
void main(){
   clrscr();
   A *aptr, a1;
   B b1;
   aptr = &b1;
   aptr->showA();

   getch();
}
```


Pointers in Derived Classes

Virtual Function

- As Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms.
- An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes.
- We use the pointer to base class to refer to all the derived objects.
- But, we just discovered that base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class.
- The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer.
- How do we then achieve polymorphism?
- It is achieved using what is known as **"virtual function"**.

```
class A{
   int x, y;
   void getA(int a, int b){
      x = a;
      y = b;
   }
   public:
      virtual void show(){
         getA(10, 20);
         cout<<"\nx = "<<x;
         cout<<"\ny = "<<y;
      }
};
```

```
class B : public A{
   int a, b;
   public:
      void getB(int x, int y){
         a = x;
         b = y;
      }
      void show(){
         getB(30, 40);
         cout<<"\na = "<<a;
         cout<<"\nb = "<<b;
      }
};
```

```
void main(){
   clrscr();
   A *aptr, a1;
   B b1;
   aptr = &a1;
   aptr->show();
   aptr = &b1;
   aptr->show();
   getch();
}
```


Virtual Function

Rules for Virtual Function

- When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements :

1. The virtual function must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be use.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with same name have different prototypes, C++ considers them as overloaded function, and the virtual function mechanism is ignored.

Rules for Virtual Function

7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure Virtual Function

- It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes.
- The function inside the base class is seldom used for performing any task.
- It only serves as a placeholder.
- Exa, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'.
- Such function are **called "do-nothing"** function
- A "do-nothing" function may be defined as follows :

  virtual void display() = 0;
- Such function are called **pure virtual functions**. A pure virtual function is a function declared in a base class that has no definition relative to the base class.
- In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function.

Pure Virtual Function

- Remember that a class containing pure virtual functions cannot be used to declare any objects of its own.
- Such classes are called **abstract classes**.
- The main objective of an abstract class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

```
class A{
  public:
    virtual void display()=0;
};
```

```
class B : public A{
    int b1, b2;
  public:
    void setB(int x, int y){
      b1 = x; b2 = y;
    }
    void display(){
      cout<<"\nB1 = "<<b1;
      cout<<"\nB2 = "<<b2;
    }
};
```

```
class C : public A{
    int c1, c2;
  public:
    void setC(int x, int y){
      c1 = x; c2 = y;
    }
    void display(){
      cout<<"\nC1 = "<<c1;
      cout<<"\nC2 = "<<c2;
    }
};
```

Abstract Class

```
void main(){
  clrscr();
  B b1;
  b1.setB(30, 40);
  b1.display();
  C c1;
  c1.setC(50, 60);
  c1.display();
  getch();
}
```

Virtual Constructor

- “A constructor can not be virtual”.
- There are some valid reason that justify this statement.

  First, to create an object the constructor of the class must be of the same type as the class. But, this is not possible with a virtual implemented constructor.

  Second, at the time of calling a constructor, the virtual table could not have been created to resolve any virtual function calls.
- As a result, it is not possible to declare a constructor as virtual.

Virtual Destructor

- A virtual destructor however, is pretty much feasible in C++.
- In fact, its use is often promoted in certain situations. One such situation is when we need to make sure that the different destructors in an inheritance hierarchy are called in order, particularly when the base class pointer is referring to a derived type object.
- It must be noted here that the order of calling of destructors in an inheritance hierarchy is opposite to that of constructors.

```
class A{
   public:
      ~A(){
         cout<<"\nBase class destructor";
      }
};
```

```
class B : public A{
   public:
   ~B(){
         cout<<"\nDerived class destructor";
   }
};
```

```
void main(){
   clrscr();
   A *P1;
   B B1;
   P1=&B1;

   delete P1;
   getch();
}
```

Virtual Destructor

Virtual Destructor

- Here both base class A and the derived class B have their own destructors.
- Now, an A class pointer has been allocated a B class object. When this object pointer is deleted using the delete operator, it will trigger the base class destructor and the derived class destructor won't be called at all.
- This may lead to a memory leak situation.
- To make sure that the derived class destructor is mandatorily called, we must declare the base class destructor as virtual as follow :

```
virtual ~A(){
        cout<<"\nBase class destructor";
}
```

Thank you