Inheritance

Extending Classes

Introduction

- Reusability is one of the most powerful feature of OOP.
- It is always nice if we could reuse something that already exists rather than trying to create the same all over again.
- It would not only save time and money but also reduce frustration and increase reliability.
- **For instance**, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

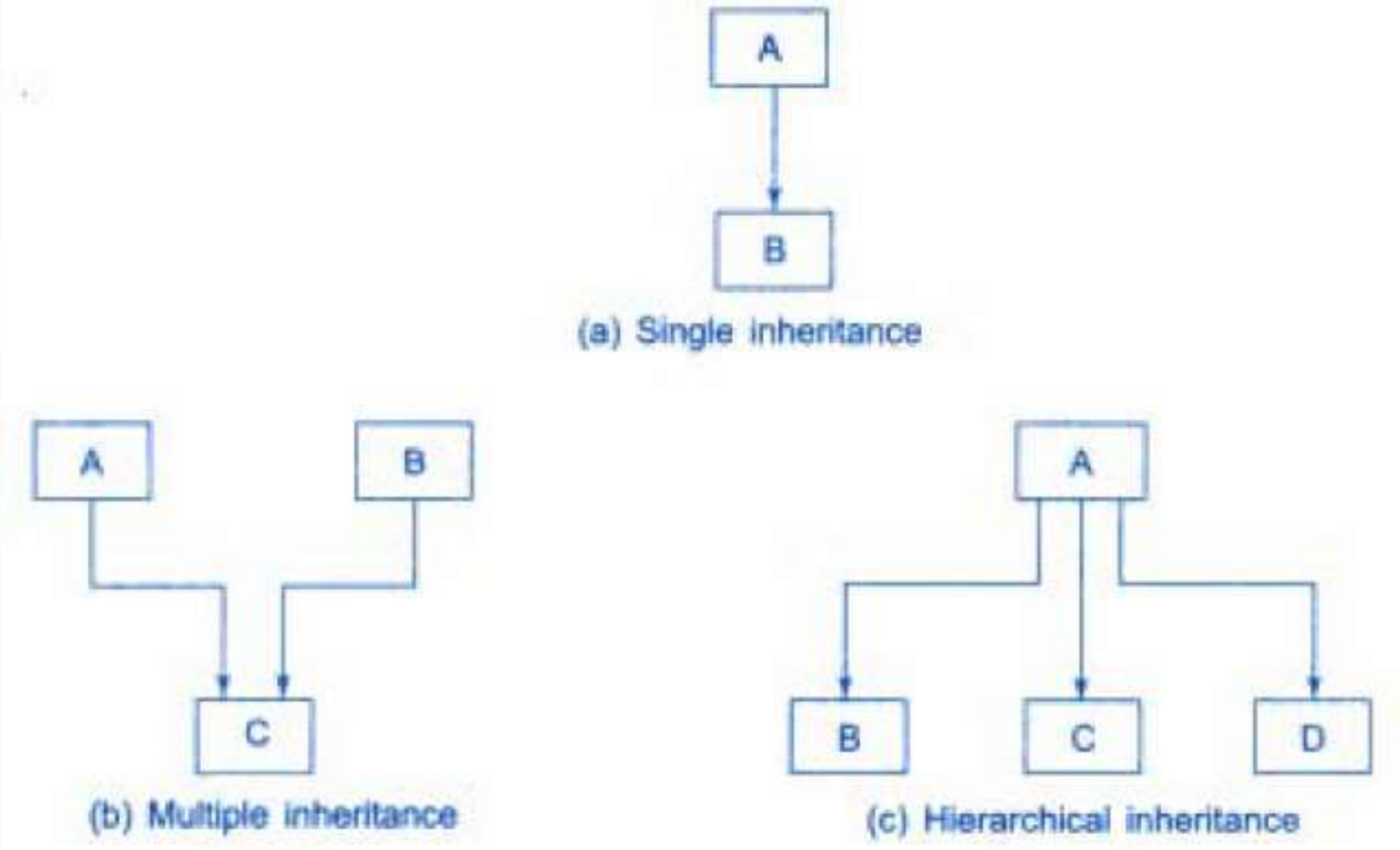
Introduction

- C++ strongly supports the concept of *reusability.*
- Once a class has been written and tested, it can be adapted by other programmers to suit their requirements.
- This is done by creating new classes & reusing the properties of the existing ones.
- The mechanism of deriving a new class from an old one 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

- ***Single inheritance*** **-** A derived class with only one base class.
- ***Multilevel inheritance*** **-** The mechanism of deriving a class from another 'derived class’.
- ***Multiple inheritance*** **-** One derived class with several base classes.
- ***Hierarchical inheritance*** **-** The traits of one class may be inherited by more than one class
- ***Hybrid inheritance*** **–** The blend of two or more types of inheritance

Various Forms of Inheritance

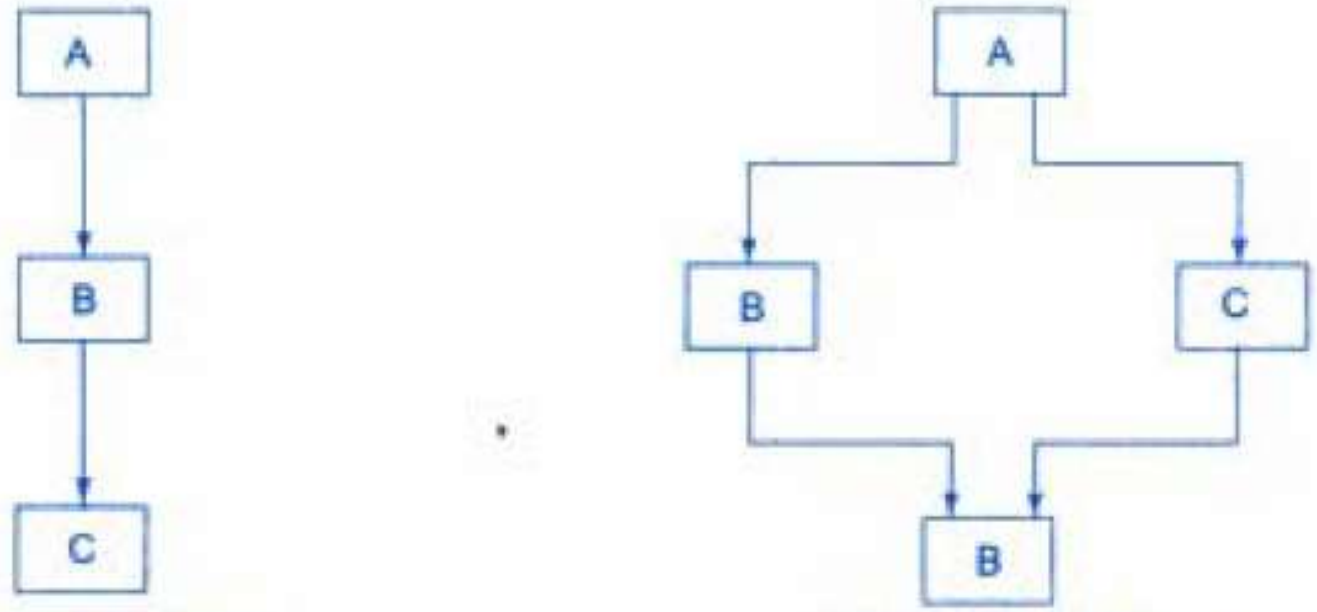


(a) Single inheritance

(b) Multiple inheritance

(c) Hierarchical inheritance

Various Forms of Inheritance



(d) Multilevel inheritance

(e) Hybrid inheritance

Defining Derived Classes

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

Syntax:

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

- The colon indicates that *derived-class-name* is derived from the *base-class-name.*
- The *usability-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 *privately derived or publicly derived.*

Examples

```
class ABC: private XYZ              // private derivation
{
        members of ABC
};

class ABC: public XYZ               // public derivation
{
        members of ABC
};

class ABC: XYZ                       // private derivation by default
{
        members of ABC
};
```

Defining Derived Classes

- When a base class is ***privately inherited*** by a derived class, 'public members' of the base class become 'private members' of the derived class

- So, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

- Remember, a public member of a class can be accessed by its own objects using the ***dot operator***.

- Hence, no member of the base class is accessible to the objects of the derived class.

Defining Derived Classes (Contd…)

- Similarly, when the base class is ***publicly inherited***, 'public members' of the base class become “***public members***” of the derived class.
- Therefore, they are accessible to the objects of the derived class.
- In both the cases, the private members are not inherited
  - Thus, the private members of a base class will never become the members of its derived class.

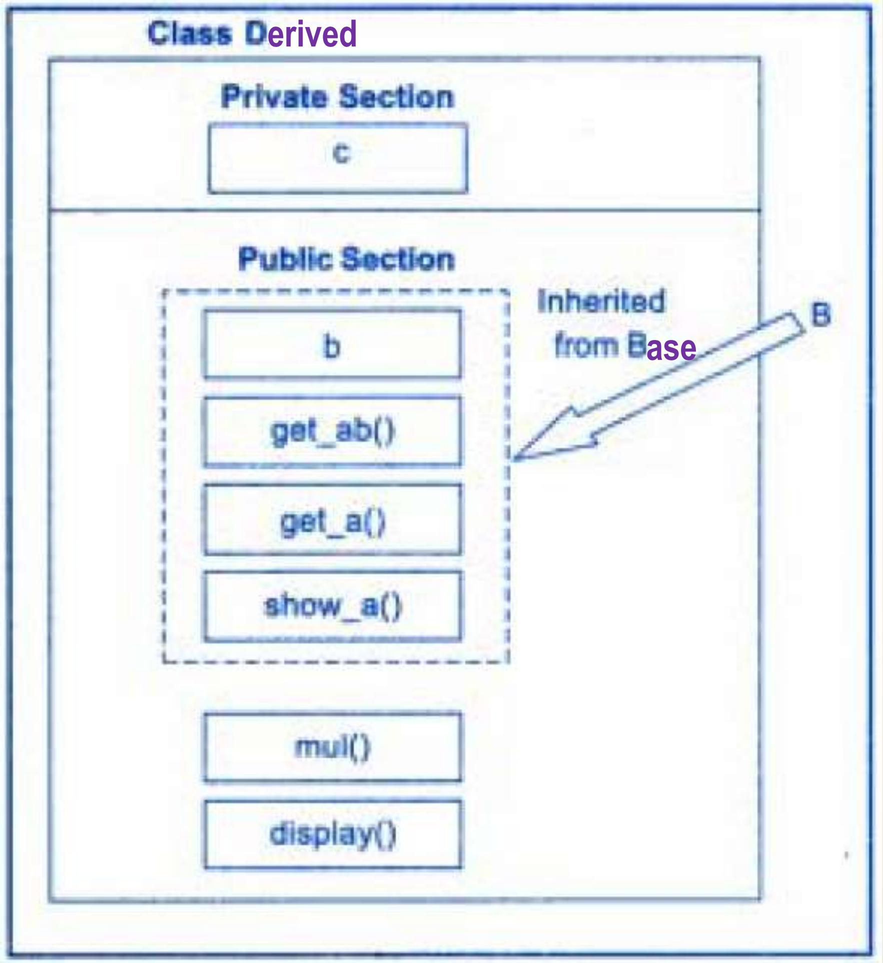
Single Inheritance: Public Derivation

```
class Base
{               int a;
        public:
                int b;
                void get_ab()
                {       a=5;  b=10;         }

                int get_a()
                {       return a;           }

                void show_a()
                {       cout<<“\na = ”<<a;      }
};
class Derived : public Base             // public derivation
{
                int c;
        public.
                void multi()
                {       c = b*get_a();      }

                void display()
                {       cout<<“\na = ”<<get_a();
                         cout<<“\nb = ”<<b;
                         cout<<“\nc = ”<<c;
                }
};
```

```
int main()
{
        Derived d;

        d.get ab(};
        d.multi(};

        d.show_a();
        d.display();

        d.b = 20;
        d.multi();
        d.displ ay();

        return 0;
}
```

Adding More Members to A Class

By Public Derivation



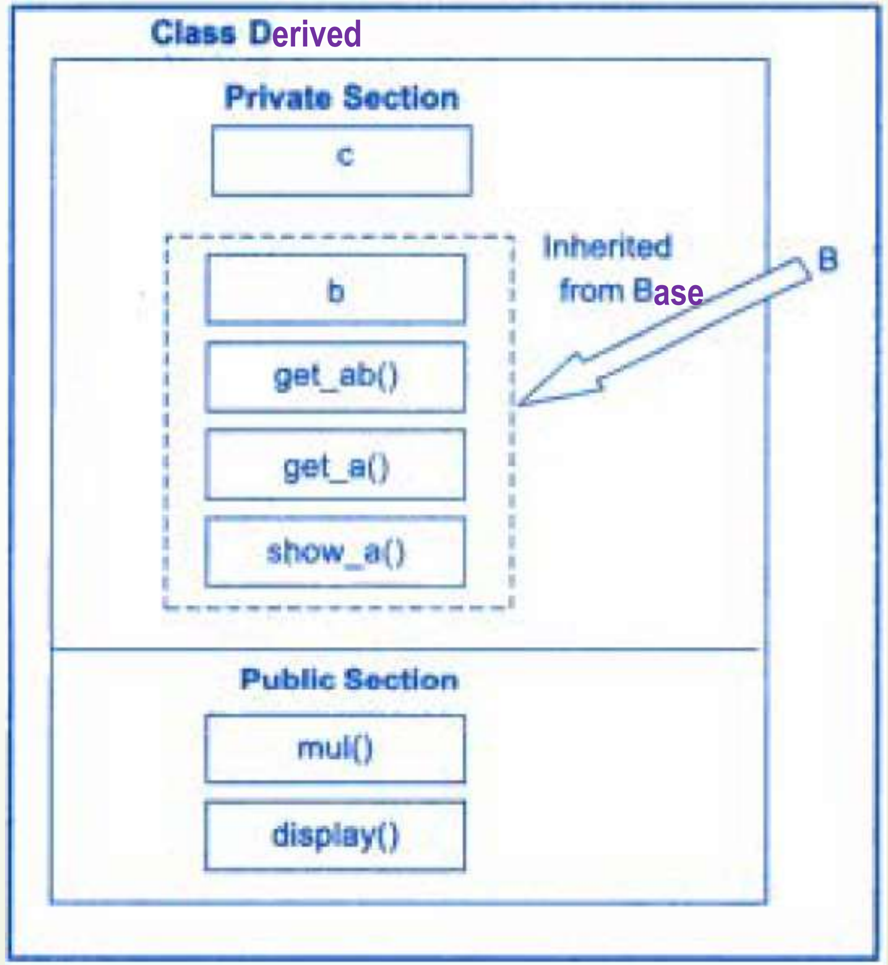
Single Inheritance: Private Derivation

```
class Base
{               int a;                          // Not inheritable
        public:
                int b;
                void get_ab()
                {       a=5;  b=10;         }
                int get_a()
                {       return a;           }
                void show_a()
                {       cout<<“\na = ”<<a;      }
};
class Derived : private Base            // private derivation
{
                int c;
        public.
                void multi()
                {       get_ab();
                        c = b*get_a();      }
                void display()
                {       show_a();
                        cout<<“\nb = ”<<b;
                        cout<<“\nc = ”<<c;
                }
};
```

```
int main()
{
        Derived d;

        // d.get ab();      will not work
        d.multi(};

        // d.show_a();      will not work
        d.display();

        // d.b = 20;        will not work
        d.multi();
        d.displ ay();

        return 0;
}
```

Adding More Members to A Class

By Private Derivation



Single Inheritance: Private Derivation

The statements such as

```
d.get_ab();          // get_ab() is private
d.get_a();
d.show_a() ;
```

will not work.

However, these functions can be used inside mul() and display() like normal functions as:

```
void mul()
{
        get_ab();
        c = b * get_a();
}
```

```
void display()
{
        show_a();
        cout <<“b = ” << b;
        cout<<“c = ” <<c;
}
```

Making a Private Member Inheritable

- A private member of a base class cannot be inherited. What to do if the private data needs to be inherited by a derived class?
- This can be accomplished by changing the visibility mode from private to public.
- This would make it accessible to all the other functions of the program, thus taking away the advantage of ***data hiding***.
- C++ provides a third visibility modifier, ***protected***, which serve the limited purpose.
- A member declared as ***protected*** is accessible by member functions within its class and any class immediately derived from it and not accessible to the outside functions.

Making a Private Member Inheritable

- When a ***protected*** member is inherited in ***public*** mode, it becomes protected in the derived class too and therefore is accessible by the member functions of the derived class.

- It is also ready for further inheritance.

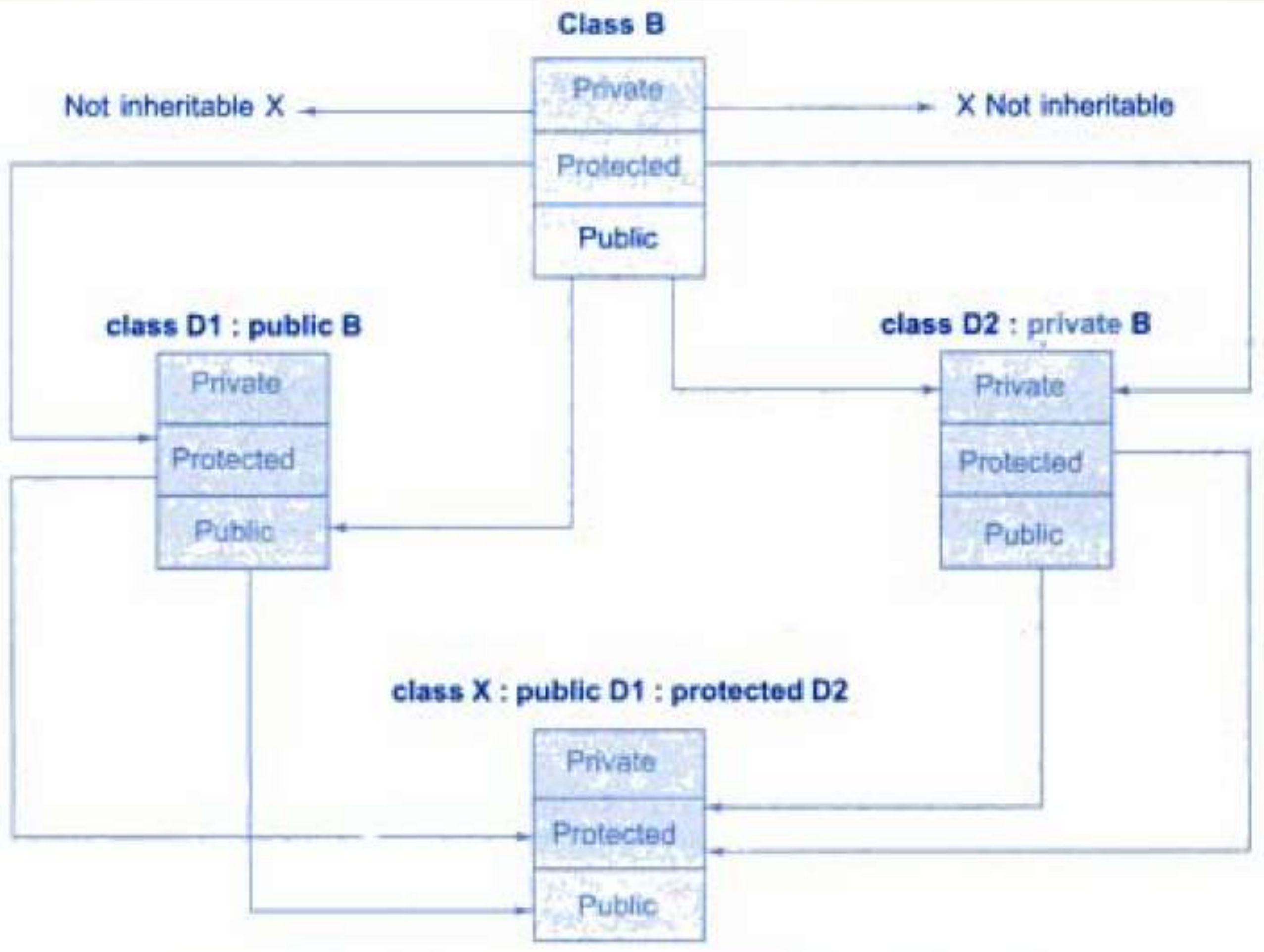
- A ***protected*** member inherited in the ***private*** mode, becomes ***private*** in the derived class.

- Although it is available to the member functions of the derived class, it is not available for further inheritance.

Class with All Visibility Modes

```
class alpha
{
        private:                     // optional
                --------             // visible to member functions
                --------             // within its class
        protected:
                --------             // visible to member functions
                --------             // of its own and derived class
        public:
                 --------            // visible to all functions
                 --------            // in the program
};
```

Effect of Inheritance on Visibility of Members



Visibility Modes

The keywords ***private, protected*** and ***public*** may appear in any order and any number of times in the declaration of a class.

For example,

```
class beta
{
        protected:
                        -----------
        public:
                        -----------
        private:
                        -----------
        public:
                        -----------
};                                              // is a valid class definition.
```

Visibility Modes

However, the normal practice is to use them as follows:

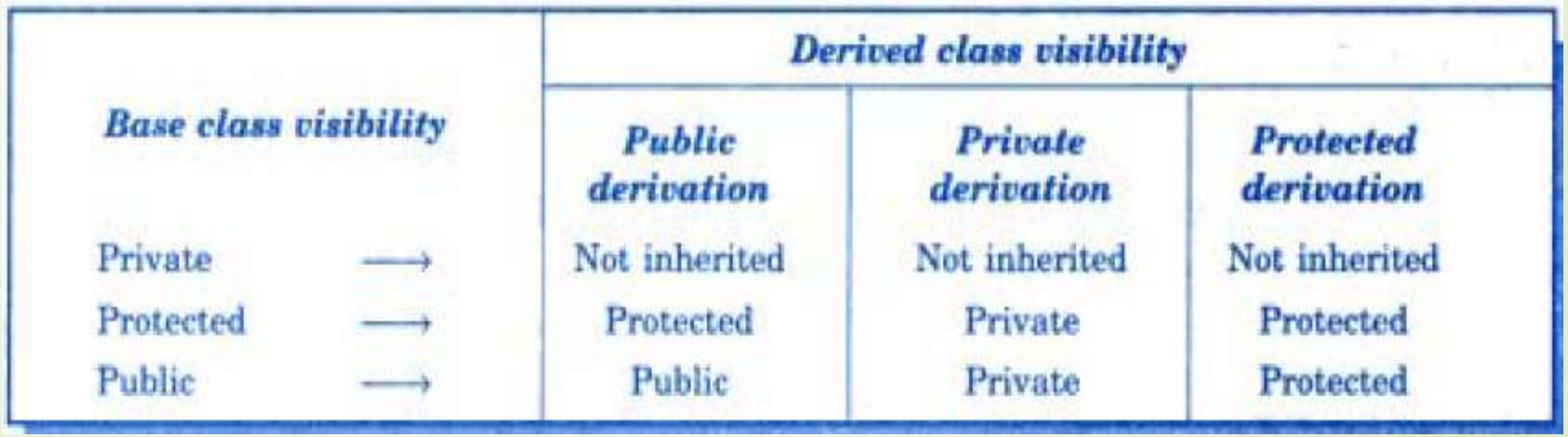
For example,

```
class beta
{
                -----------          // private by default
                -----------
     protected:
                -----------
                -----------
     public:
                -----------
                -----------
};
```

Visibility of Inherited Members

- It is also possible to inherit a base class in protected mode (known as ***protected derivation***)
- In this, both the public and protected members of the base class become protected members of the derived class.

Visibility of Base Class Members in Different Types of Derivation



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

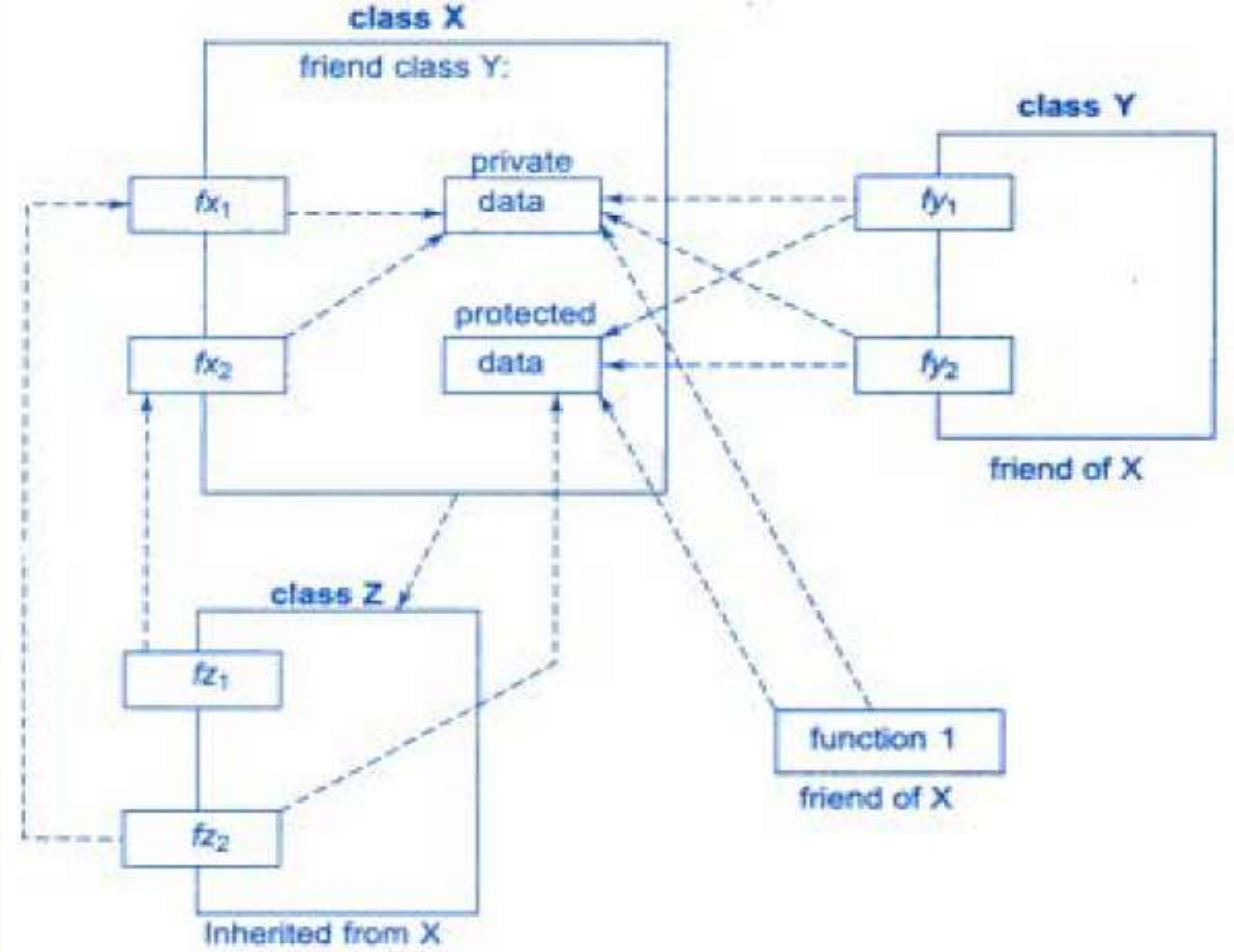
Access Control

- What are the various functions that can have access to the private and protected members of a class?
- They could be:

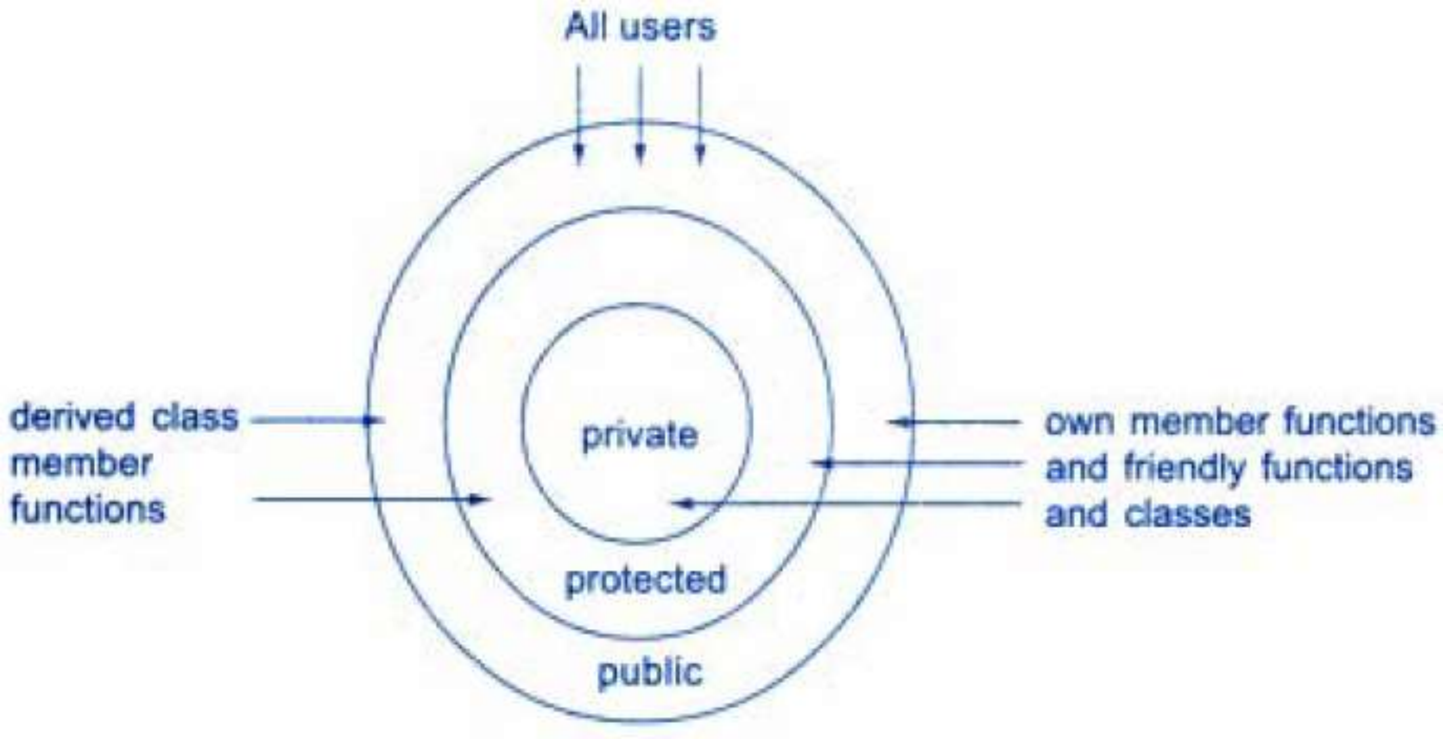
  1. A function that is a friend of the class.
  2. A member function of a class that is a friend of the other class.
  3. A member function of a derived class.

- However, they can access the private data through the member functions of the base class.

How Access Control Works?

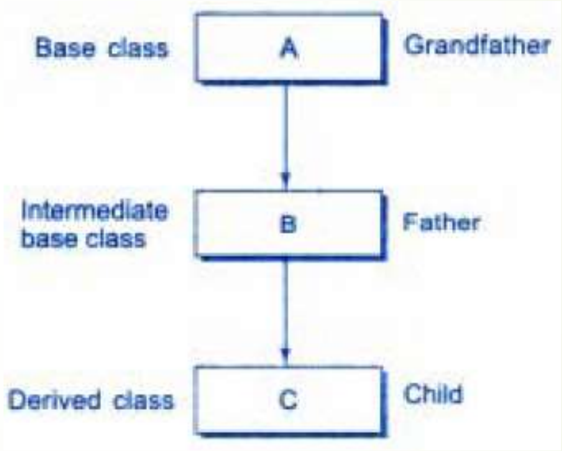


A Simplified View of Access Control


All users
derived class member functions
private
protected
public
own member functions and friendly functions and classes

Multilevel Inheritance

- When a class is derived from another derived class, is known as ***Multilevel Inheritance***. **For example**,



- Class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.
- Class B is known as ***intermediate* base class** since it provides a link for the inheritance between A and C.
- The chain ABC is known as ***inheritance path***.

Multilevel Inheritance

- **A derived class with multilevel inheritance is declared as follows:**

```
class A{ ------ } ;                 // Base class
class B: public A { ------ };       // B derived from A
class C: public B { ------ };       // C derived from B
```

- This process can be extended to any number of levels.

- **For example,** Assume that the test results of a batch of students are stored in three different classes.

  Class ***student*** stores the roll_number.

  Class ***test*** stores the marks obtained in two subjects and

  Class ***result*** contains the total marks obtained in the test.

- The class ***result*** can inherit the details of the marks obtained in the test and the rollno of students through ***multilevel inheritance***.

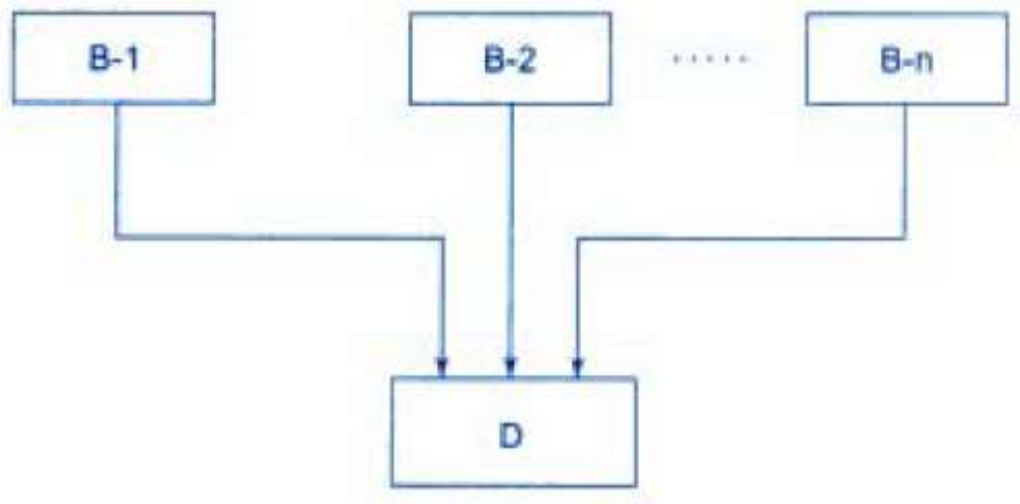
Program Example

```
class student
{
        protected:  int rollno:
        public:
                void get_no(int a)
                {     rollno = a;      }

                void put_no();
                {    cout <<“Roll Number: ”<< rollno;  }
};
class test : public student                    // 1st level derivation
{
        protected:  float sub1, sub2;
        public:
                void get_marks (float x, float y)
                {       subl = x:
                        sub2 = y;
                }
                void put_marks ()
                {       cout <<“Marks in Subl =”<<sub1;
                        cout <<“Marks in Sub2 ="<<sub2;
                }
};
```

```
class result : public test      // 2nd level derivation
{
                float total;
        public:
                void display()
                {
                        total = sub1 + sub2;
                        put_no();
                        put_marks();
                        cout<<“Total = ”<<total;
                }
};

int main()
{
                result s1;

                s1.get_no(101);
                s1.get_marks(85.5, 56.15);
                sl.display();

                return 0;
}
```

Multiple Inheritance

- When a class inherits attributes of two or more classes, it is known as ***Multiple Inheritance***
- **Multiple inheritance** 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 & the intelligence of another



Multiple Inheritance

- ***Syntax*** of a derived class with multiple base classes is as follows:

```
class D: visibility B1, visibility B2, …
{
        -------
        -------              (Body of D)
        -------
};
```

- Where *visibility* may be either ***public*** or ***private*** and the base classes are separated by commas.

Example:

```
class Derived : public Base1, public Base2
{
    public:
        void show()
        {    ------     }
};
```

Program to Demonstrate Multiple Inheritance

```
class Base1
{
        protected:
                int b1;
        public:
                void get_b1(int x)
                {
                        b1 = x;
                }
};
class Base2
{
        protected:
                int b2;
        public:
                void get_b2(int y)
                {
                        b2 = y;
                }
};
```

```
class Derived : public Base1, public Base2
{
        public:
                void display()
                {
                        cout<<"b1 = "<<b1<<endl;
                        cout <<"b2 = "<<b2<<endl;
                        cout <<"b1*b2 = "<<b1*b2;
                }
};
int main()
{
        Derived D;

        D.get_b1(10) ;
        D.get_b2(20) ;
        D.display() ;

        return 0;
}
```

Ambiguity Problem in Multiple Inheritance

- Ambiguity problem may arise when a function with the same name appears in more than one base class. Compiler may get confuse which function to invoke. **For Example:**

```
class Base1
{
        protected:
                int b1;
        public:
                void get_b1(int x)
                {       b1 = x;       }

                void show()
                {
                        cout<<“b1 = “<<b1;

                }
};
```

```
class Base2
{
        protected:
                int b2;
        public:
                void get_b2(int y)
                {       b2 = y;       }

                void show()
                {
                        cout<<“b2 = ”<<b2;
                }
};
```

- Which show() function is used by the derived class when we inherit these two classes?

Solution to Ambiguity Problem

- Ambiguity problem can be solved by using ***Scope Resolution operator***.

For Example:

```
class Derived : public Base1, public Base2
{
        public:
                void display()
                {
                        Base1 :: show();
                        Base2 :: show();
                        cout <<"b1*b2 = "<<b1*b2;
                }
};
```

Program to Deal with Ambiguity Problem

```
class Base1
{
        protected:
                int b1;
        public:
                void get_b1(int x)
                {       b1 = x;       }
                void show()
                {
                        cout<<"b1 = "<<b1;
                }
};
class Base2
{
        protected:
                int b2;
        public:
                void get_b2(int y)
                {       b2 = y;       }
                void show()
                {
                        cout<<"b2 = "<<b2;
                }
};
```

```
class Derived : public Base1, public Base2
{
        public:
                void display()
                {
                        Base1 :: show();
                        Base2 :: show();
                        cout <<"b1*b2 = "<<b1*b2;
                }
};
int main()
{
        Derived D;
        D.get_b1(10) ;
        D.get_b2(20) ;
        D.display() ;

        return 0;
}
```

Ambiguity Problem in Single Inheritance

- Ambiguity may also arise in single inheritance applications.

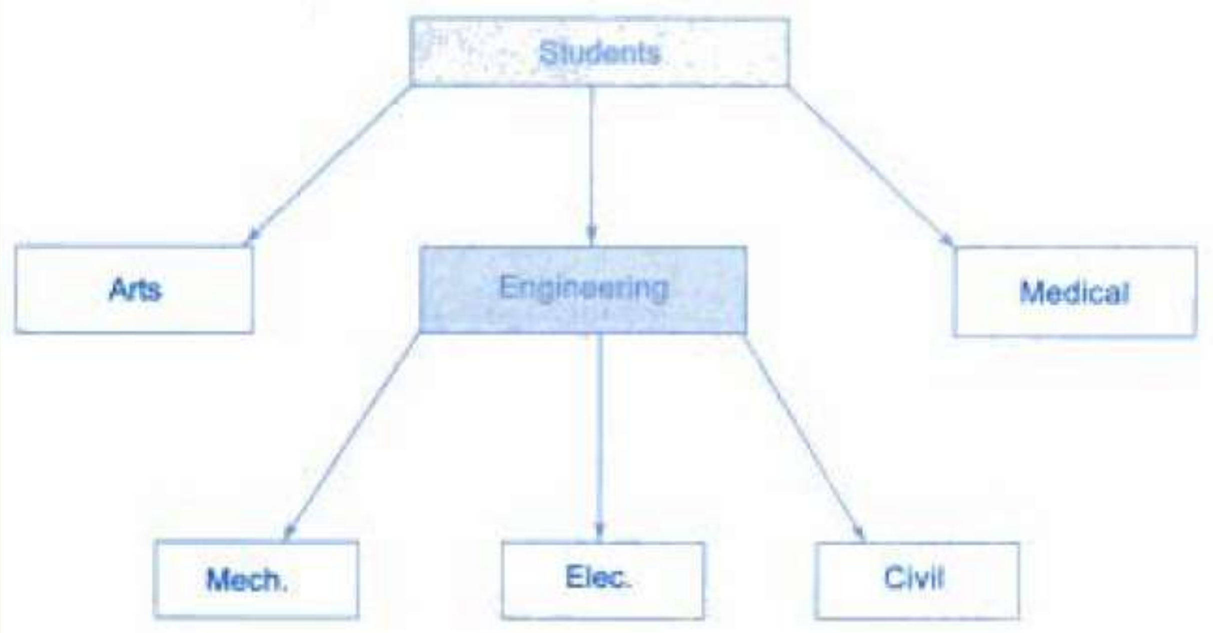
For instance:

```
class Base
{
        protected:
                int b;
        public:
                void get(int x)
                {
                        b = x;
                }
                void show()
                {
                        cout<<“\nB = “<<b;

                }
};
```

```
class Derived : public Base
{
        public:
                void show()
                {
                        cout <<“\nB Square = "<<b*b;
                }
};
int main()
{
        Derived D;
        D.get(10);
        D.show();               // Invokes show() of Derived
        D.Base::show();         // Invokes show() of Base
}
```

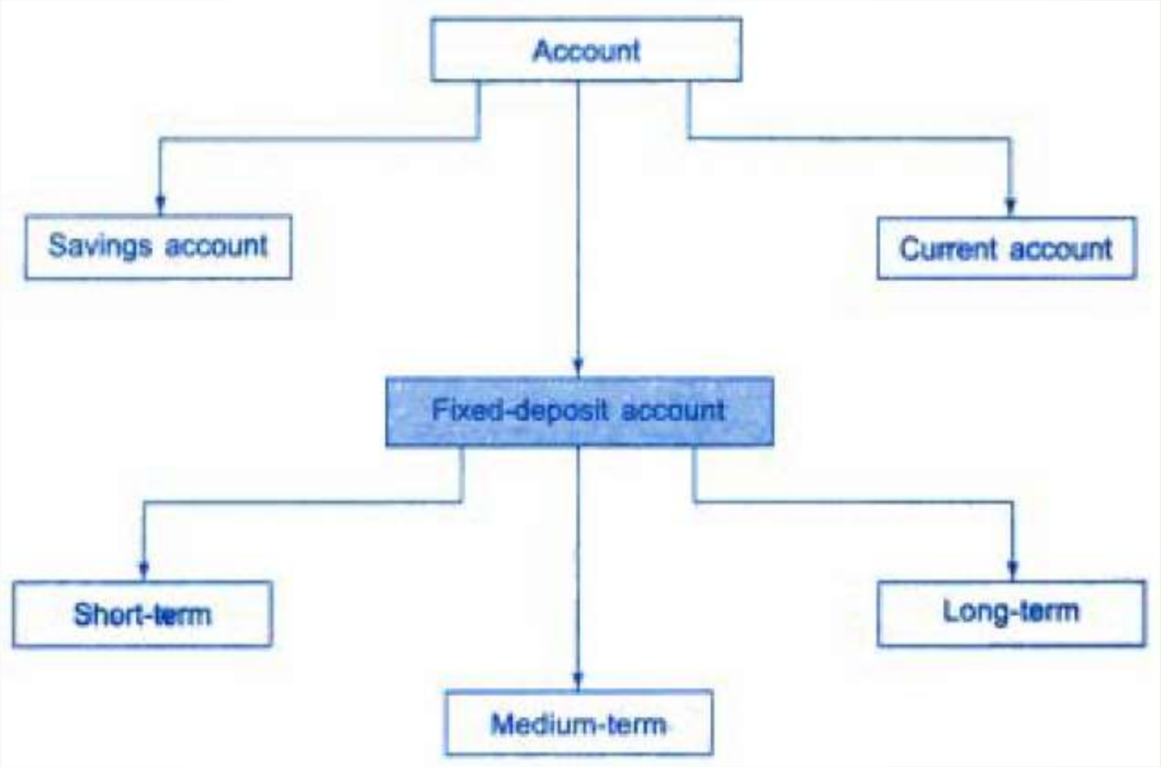
Hierarchical Inheritance

- When many classes inherit attributes of a single base class, it is known as ***Hierarchica1 Inheritance***
- Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level. **For Example:**



Hierarchical classification of students

Classification of Bank Accounts



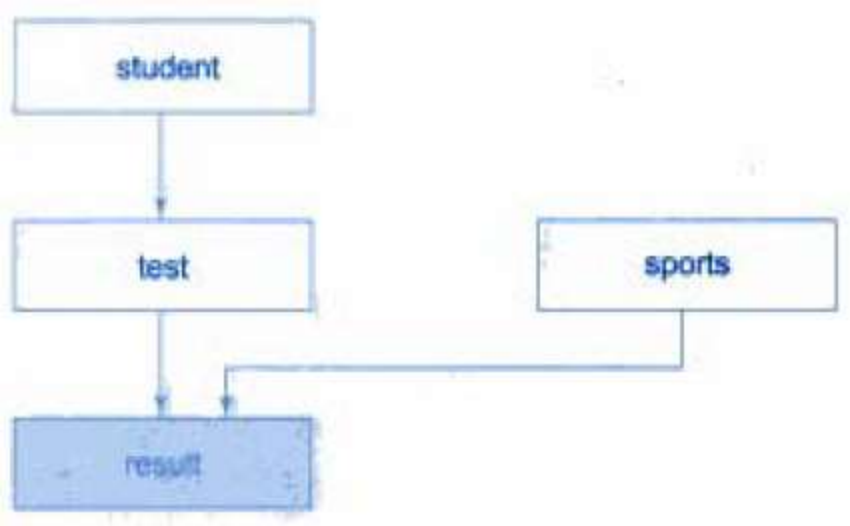
- Such problems can be easily converted into **class hierarchies**.
- ***Base class*** includes all the features that are common to the subclasses.
- ***Subclass*** can further serve as a base class for the lower level classes and so on.

Hybrid Inheritance

- There may be situations where we need to apply two or more types of inheritance to design a program.

For instance:

- Consider the case of processing the ***student result***. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called ***sports***.



Hybrid Inheritance: ***Multilevel & Multiple Inheritance***

Program to Demonstrate Hybrid Inheritance

```
class student
{
        protected:
                int rollno;
        public:
                void getno(int a)
                {       rollno = a;       }
                void putno()
                {       cout<<"Roll No: "<<rollno;  }
};
class test : public student
{
        protected:
                float sub1, sub2;
        public:
                void getmarks(float m1, float m2)
                {       sub1 = m1;
                        sub2 = m2;
                }
                void putmarks()
                {       cout<<"\nMarks obtained: ";
                        cout<<"\nSubl = "<<sub1;
                        cout<<"\nSub2 = "<<sub2;
                }
};
```

```
class sports
{
        protected:
                float score;
        public:
                void getscore(float s)
                {       score = s;      }
                void putscore()
                {
                        cout<<"\nSports Weight: "<<score;
                }
};
class result : public test, public sports
{
                float total;
        public:
                void display()
                {       total = sub1 + sub2 + score;
                        putno();
                        putmarks();
                        putscore();
                        cout<<"\nTotal Score: "<<total;
                }
};
```

```
int main()
{
      result s1;
      s1.getno(1408);
      s1.getmarks(28.5, 34.0);
      s1.getscore(8.0);
      s1.display();
}
```

Thanks