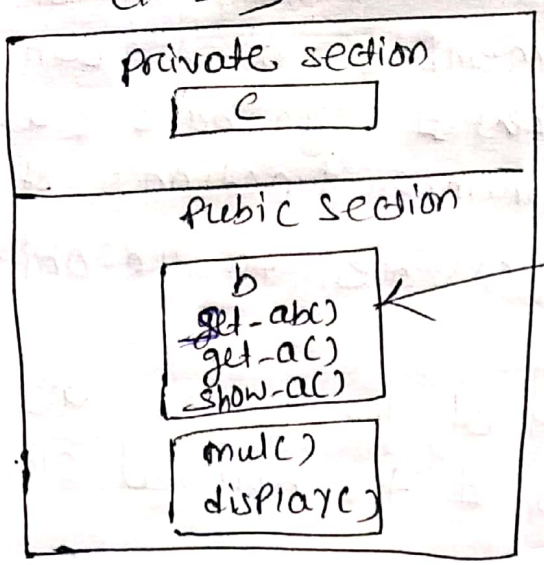
presents in Class D

presents in Class B

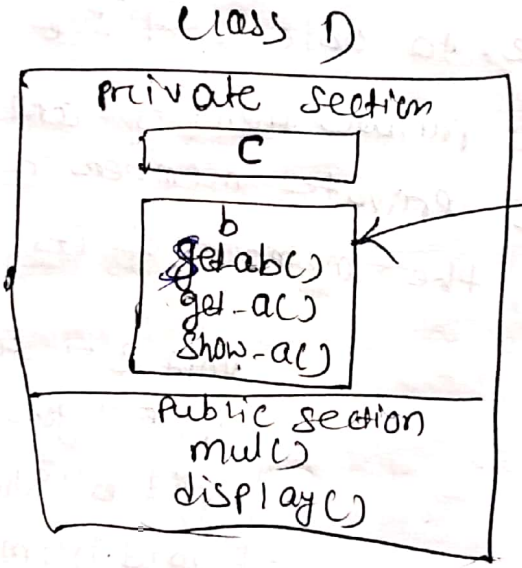
```
void mul()
{
   c= b* get_a();
}
```



If

class D : private B

then




```
d.setabc();  // won't work
d.get_a();   // won't work
d.show_a();  // won't work
```

because of private inheritance

Base class member access specifier	Types of inheritance		
	Public inheritance	Protected inheritance	Private inheritance
Public	Public in derived class can be access directly by member fun, friend fun, non member fun	protected in derived class can be access directly by member fun and friend fun	Private in derive class can be access directly by member fun and friend fun
Protected	Protected in derived class	protected in derived class	Private in derived class
Private	Hidden in derived class	Hidden in derived class	Hidden in derived class

Making a private member inheritable

→ Let private data need to be inherited then we have to make its public. (by modifying the visibility limit of the private member by making it public).

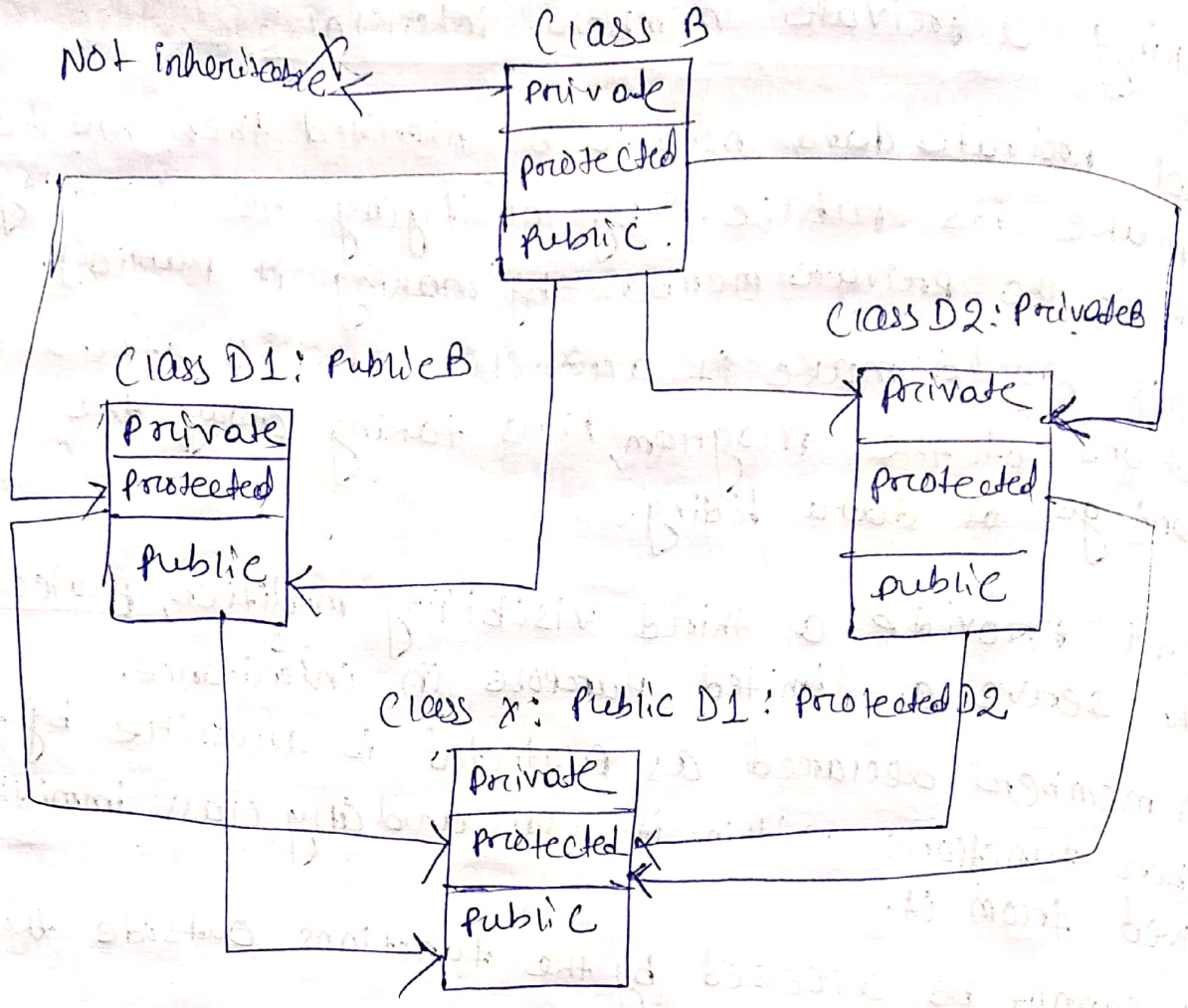
→ This would make it accessible to all other functions of the program, thus taking away the advantage of data hiding.

→ C++ provide 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 immidiately derived from it.

→ It cannot be accessed by the functions outside these two classes.

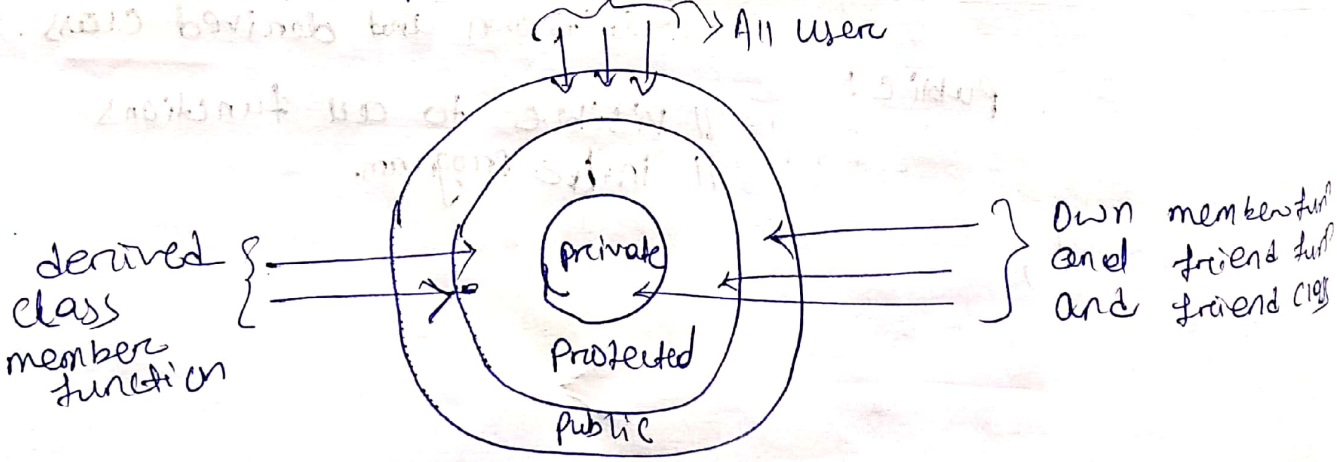
```
Class ABC
{
    private:
        ....        // visible to member functions
        ....        // within its class.

    protected:
        ....        // visible to member functions
        ....        // of its own and derived class.

    public:
        ....        // visible to all functions
        ....        // in the program.

};
```



→ A function that is friend of the class can access both private and protected.

→ A member function of a class that is friend of the class can access both private and protected.

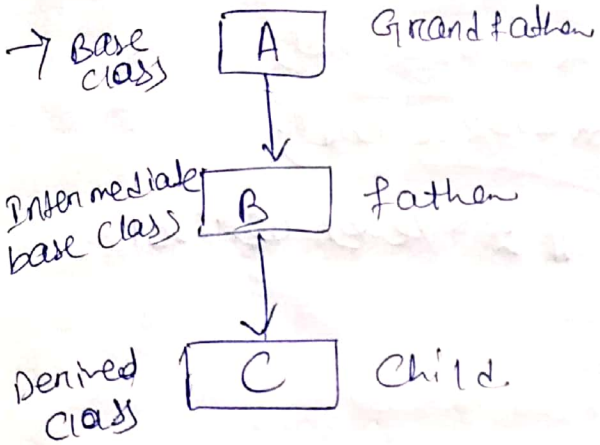
→ A member function of a derived class only access the protected but not private.



Simple view of access control to the member of a class.

Multilevel inheritance :-

→ A derived class is derived from another derive class is called multilevel inheritance



→ C contains the members of A and B.

→ Last derive class acquires all the members of all its base classes.

Program of Multilevel Inheritance :-

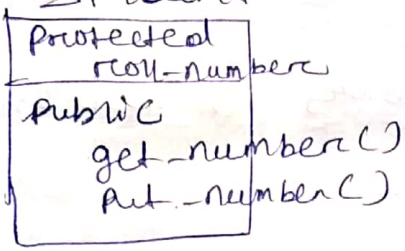
```
class student
{  protected:
        int roll_number;
   public:
        void get_number(int);
        void put_number();
};
void student :: get_number(int a)
{   roll_number = a;
}
void student :: put_number()
{ cout<< "ROll_number : " << roll_number << endl;
}
class test : public student // first level derivation
{
   protected :
        float sub1;
        float sub2;
   Public:
        void get_marks (float, float);
        void put_marks ();
};
```

```
void test :: get_marks ( float x, float y)
{ sub1 = x ; sub2 = y; }
void test :: put_marks ()
{ cout << " Marks in SUB1 = " << sub1 << endl;
  cout << " Marks in SUB2 = " << sub2 << endl;
}
Class result : Public test // 2nd level derivation
{
    float total;
    Public:
        void display();
};
void result :: display ()
{
    total = sub1 + sub2;
    Put_number ();
    Put_marks ();
    Cout << " Total = " << total << endl;
}
int main ()
{
    result student1;
    Student1 . get_number (111);
    Student1 . get_marks (75.0, 59.5);
    Student1 . display ();
    return 0;
}
```

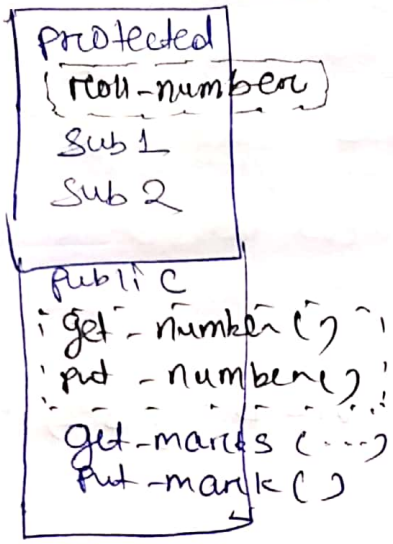
Output

```
Roll number : 111
Marks in SUB1 = 75
Marks in SUB2 = 59.5
Total = 134.5
```

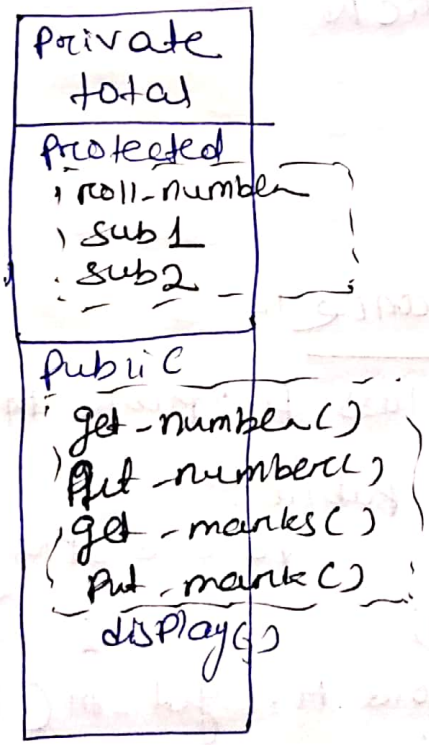
Student



test : public Student

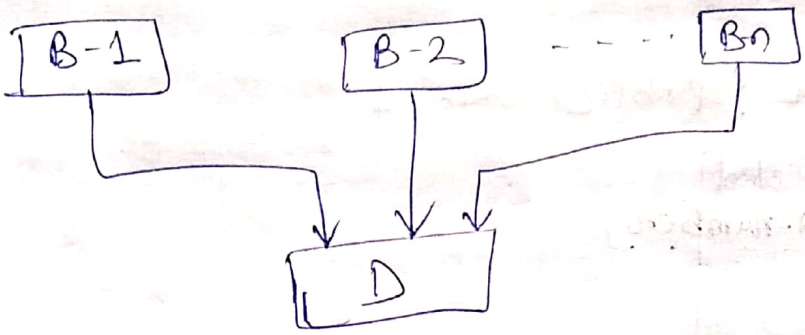


result : public test



Multiple Inheritance :-

→ A class inherit the attributes of two or more classes.



Syntax :-

```
Class D : visibility B-1, visibility B-2, ....
{

};
```

Ex :-

```
Class P : public M, public N
{
  ---
};
```

Program of Multiple Inheritance :-

```
class M
{   protected:
      int m;
    public:
      void get_m(int);
};
class N
{   protected:
      int n;
    public:
      void get_n(int);
};
class P : public M, public N
{ public:
      void display();
};
void M :: get_m(int x)
{  m=x; }
void N :: get_n(int y)
{
   n=y; }
void P :: display()
{ cout << m << n << m*n;
```

```
int main()
{
    P ob;
    ob.get_m(10);
    ob.get_n(20);
    P.display();
    return 0;
}
```

Output

10 20 200

Ambiguity Resolution in Inheritance :-

→ When a function with the same name appears in more than one base class.

Ex:-

```
class M
{ public:
    void display()
    {
        cout << "class M in";
    }
};

class N
{ public:
    void display()
    {
        cout << "class N in";
    }
};

class P : public M, public N
{ public:
    void display()
    {
        M::display();
    }
};
```

```
int main()
{
    P ob;
    ob.display();
}
```

→ To solve ambiguity we can define a named instance within the derived class, using the scope resolution operator.

→ overrides display() of M and N.

// display();

→ Ambiguity because display() is present in M and N

Function overriding :-

→ allows us to have same function in child class which is already present in the ~~the~~ Parent class. A child class inherits the data members and member functions of Parent class, but when you want to override a functionality in the Child class then you can use function overriding.

→ It is like creating a new version of an old function in the child class.

```
Class A
{
  public:
    void display()
    {
      cout<<" Class A "<<endl;
    }
};
Class B
{
  public:
    void display()
    {
      cout<<" Class B" <<endl;
    }
};
Class C : public A, public B
{ public:
    void view()
    { display(); //cout<<"
                    Class C";
};
```

```
int main()
{
  C ob;
  ob.view();
// or
  ob.display();
return 0;
}
```

A or B

Output

Error:
reference to display is ambiguous

A or B

```
Ob. A :: display();   // Ob.display();
or
Ob.B :: display();   // Ob.display();
```
in int main();

```
A :: display();   // display();
B :: display();   // display();
```
in view().

→ Ambiguity may also arise in single inheritance.

Ex :-

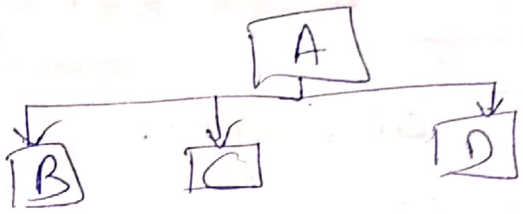
```
Class A
{ public:
    void display()
    { cout << " A \n";
    }
};
Class B : public A
{
  public:
    void display()
    {
      cout << " B \n";
    }
};
int main()
{
    B Ob;          ——→ derive class object.
    Ob.display();  → invokes display() in B.
    Ob.A :: display(); → invokes display() in A.
    Ob.B :: display(); → invokes display() in B.
    return 0;
}
```

Output

```
B
A
B
```

Hierarchical Inheritance :-

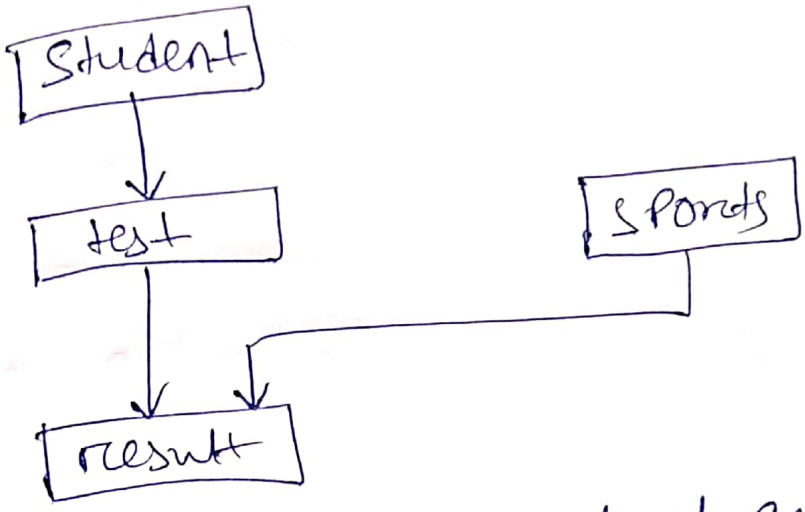
→ deriving more than one class from a base



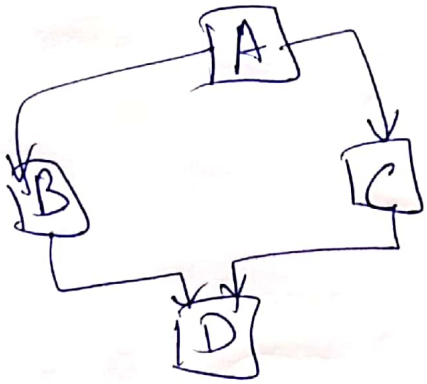

```
Class A
{
  ----
};
Class B: public A
{
  ----
};
Class C: public A
{
  ----
};
Class D: public A
{
  ----
};
```

Hybrid Inheritance :-

→ is a combination of more than one type of inheritance



(Combination of multi level and multiple)



(Combination of Hierarchical and multiple)