

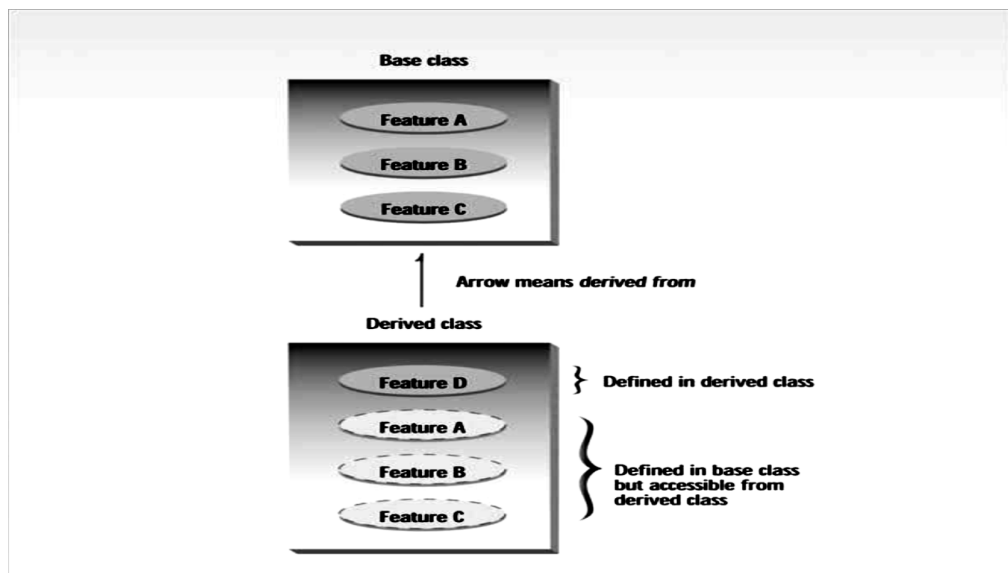
## Lecture 5

# INHERITANCE

### 5.1 Introduction

Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves. Inheritance is the process of creating new classes, called derived classes, from existing or *base classes*. The mechanism of deriving a new class from an old one is called *inheritance (or derivation)*.

The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class is unchanged by this process. The inheritance relationship is shown in Figure 5.1.



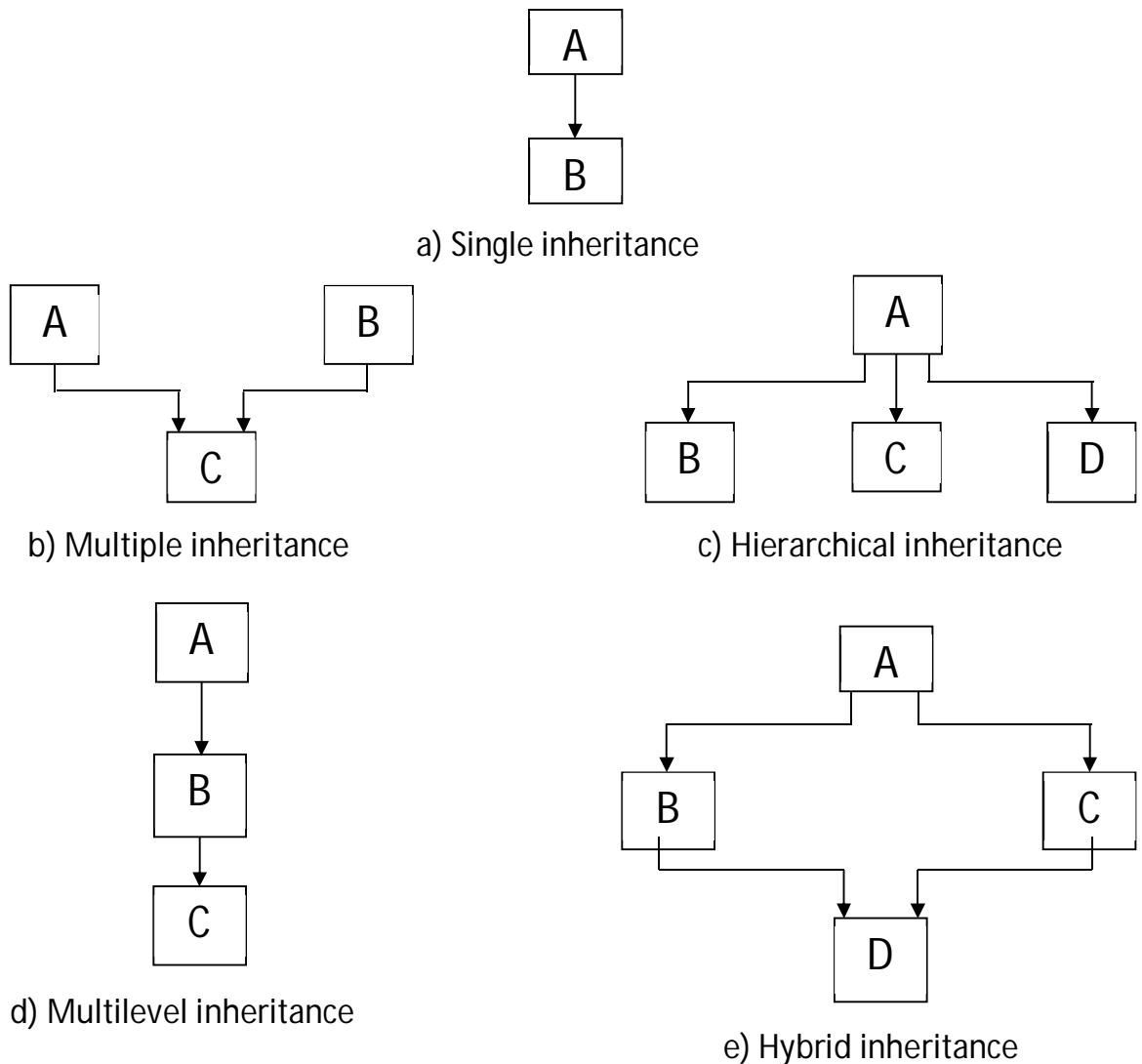
**FIGURE (5.1) Inheritance.**

Inheritance is an essential part of OOP. Reusing existing code saves time and money and increases a program's reliability.

A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Figure 5.2 shows **various form of inheritance**. The direction of arrow indicates the direction of inheritance.

- ❖ **Single inheritance**:- a derived class with only one base class.
- ❖ **Multiple inheritance**:- one derived class with several base classes.
- ❖ **Hierarchical inheritance**:- one base class may be inherited by more than one derived class.
- ❖ **Multilevel inheritance**:- The mechanism of deriving a class from another 'derived class'.



**Fig (5.2) Forms of inheritance**

## 5.2 Defining Derived Classes

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

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

The colon indicates that the *derived-class-name* is derived from the *base-class-name*. 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 *private derived* or *public derived*.

### Example:-

```
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
};
```

When a **base class** is *privately inherited* by a derived class, ‘**public members**’ of the base class become ‘**private members**’ of the **derived class** and therefore ‘public members’ of the base class can only be accessed by member functions of the derived class.

On other hand, when a base class is *publicly inherited* ‘public members’ of the base class become ‘public members’ of the derived class and therefore they are accessible to the objects of the derive class. In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derive class.

### 5.3 Single Inheritance

Let as consider a simple example to illustrate inheritance. A base class **B** and a derived class **D**.

class B contains : one private data member

one public data member

three public member functions

class D contains : one private data member and two public member functions.

**Example:-** //Single Inheritance : Public

```
#include <iostream>
```

```
using namespace std;
```

```
class B
```

```
{
```

```
    int a;    //private; not inheritable
public:
    int b;    //public ready for inheritance
    void get_ab();
    int  get_a(void);
    void show_a(void);
};
class D : public B //public derivation
{
int c;
    public:
        void mul (void);
        void display (void);
};
// function defined //
void B :: get_ab(void)
{
a = 5;   b = 10;
    }
int B :: get_a()
{
return a;
    }
void B :: show_a()
{
cout << "a = " << a << "\n";
    }
void D :: mul ()
{
```

```
c = b * get_a ();
}
void D :: display (void)
{
    cout << " a= " << get_a() << "\n";
    cout << " b= " << b << "\n";
    cout << " c= " << c << "\n";
}
// main program //
int main()
{
    D d;
    d.get_ab();
    d.mul();
    d.show_a();
    d.display();
    d.b=20;
    d.mul();
    d.display();
return 0;
}
```

The output of program:

```
a=5
a=5
b=10
c=50
a=5
b=20
c=100
```

The class **D** is a public derivation of the base class **B**. therefore, **D** inherits all public members of **B** and retains their visibility. Thus a public member of the base class **B** is also a public member of the derived class **D**. the private member of **B** cannot be inherited by **D**. the class **D**, in effect, will have more members than what it contains at the time of derivation as shown in figure 5.3 a.

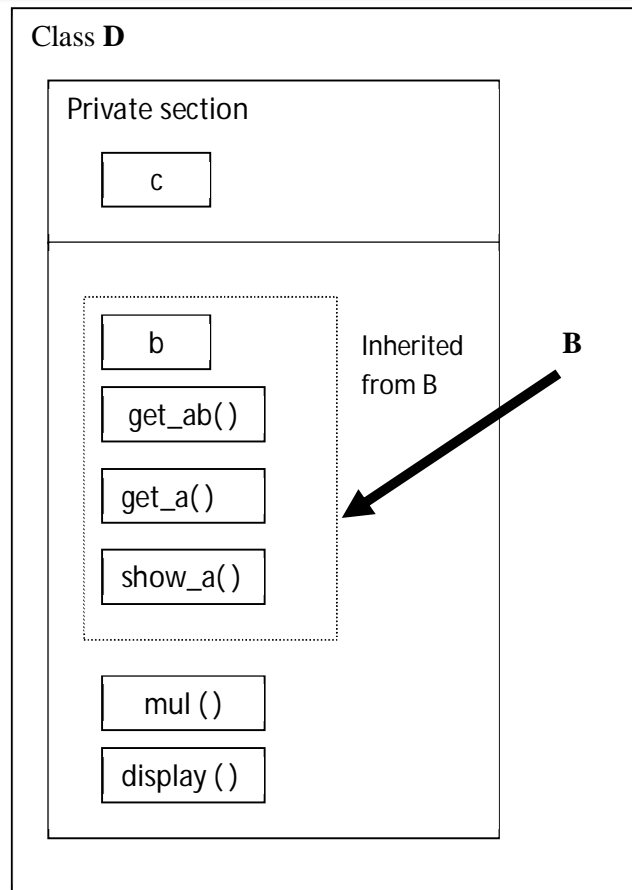


Fig 5.3 a Adding more members to a class  
(by public derivation)

Although the data member 'a' is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B**.

Let us now consider the case of *private derivation*.

```
class B
{
    int a;
public:
    int b;
    void get_ab();
    int get_a();
```

```
void show_a();  
};  
class D : private B //private derivation  
{  
    int c;  
public:  
    void mul ();  
    void display ();  
};
```

the membership of the derived class D is shown in fig. 5.3 b

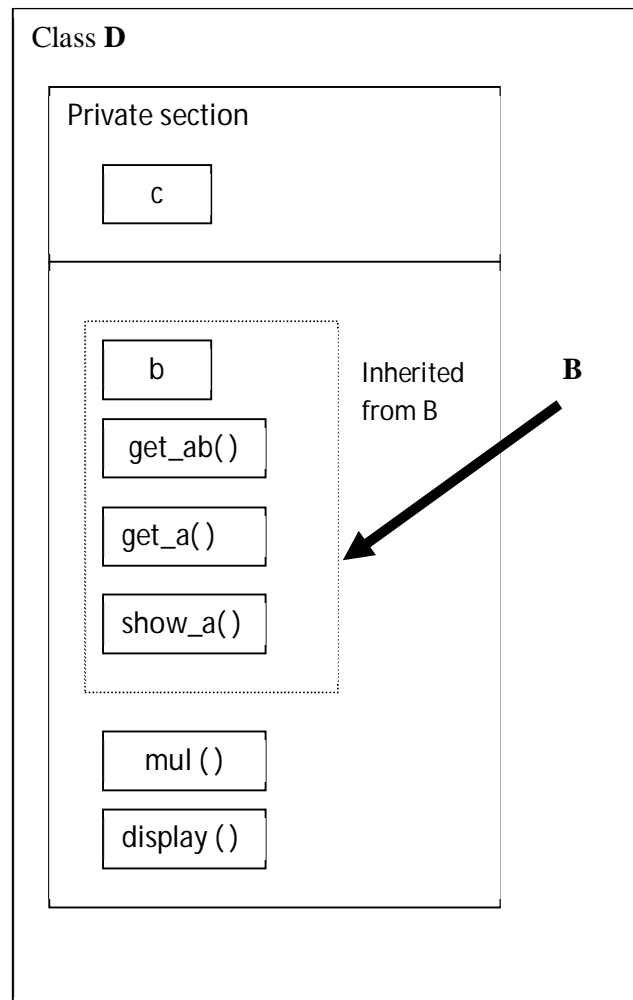


Fig 5.3 b Adding more members to a class  
(by private derivation)



In **private** derivation the public members of the base class become private members of the derived class. Therefore, the objects of **D** cannot have direct access to the public member functions of **B**.

The statements such as

```
d . get_ab( );
```

```
d . get_a();
```

```
d . show_a();
```

will not work. However, these functions can be used inside mul() and display() like the normal functions as shown below:

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

**Example:- // single inheritance : private //**

```
#include <iostream >  
using namespace std;  
class B  
{  
    int a;          //private; not inheritable  
public:  
    int b;          //public ready for inheritance  
    void get_ab( );  
    int  get_a(void);  
    void show_a(void);  
};  
class D : private B          // private derivation  
{    int c;
```

```
public:
    void mul (void);
    void display (void);
};
// function defined //
void B :: get_ab(void)
{
    cout<<"Enter the values for a and b:";
    cin>>a>>b;
}
int B :: get_a()
{
    return a;
}
void B :: show_a()
{
    cout << "a = " << a << "\n";
}
void D :: mul ()
{
    get_ab();
    c = b * get_a ();      // 'a' cannot be used directly
}
void D :: display (void)
{
    show_a();
    cout << " b= " << b << "\n";
    cout << " c= " << c << "\n";
}
```

```
// main program //
```

```
int main()
{
    D d;
    d.mul();
    d.display();
    d.b=20; // won't work; b has become private
    d.mul();
    d.display();
return 0;
}
```

The output of program:

Enter the values for a and b:5 10

a=5

b=10

c=50

Enter the values for a and b:12 20

a=12

b=20

c=240

#### 5.4 The protected Access Specifier

A private member of a base class cannot be inherited and therefore it is not available for the derived class directly. To solve this problem, by modifying the visibility limit of the private member by making it public. This would make it accessible to all the other functions of the program, thus eliminate the advantage of data hiding.

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 cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

Class alpha

```
{
    private           // optional
    .....           // visible to member functions
    .....           // within its class
```

**protected :**

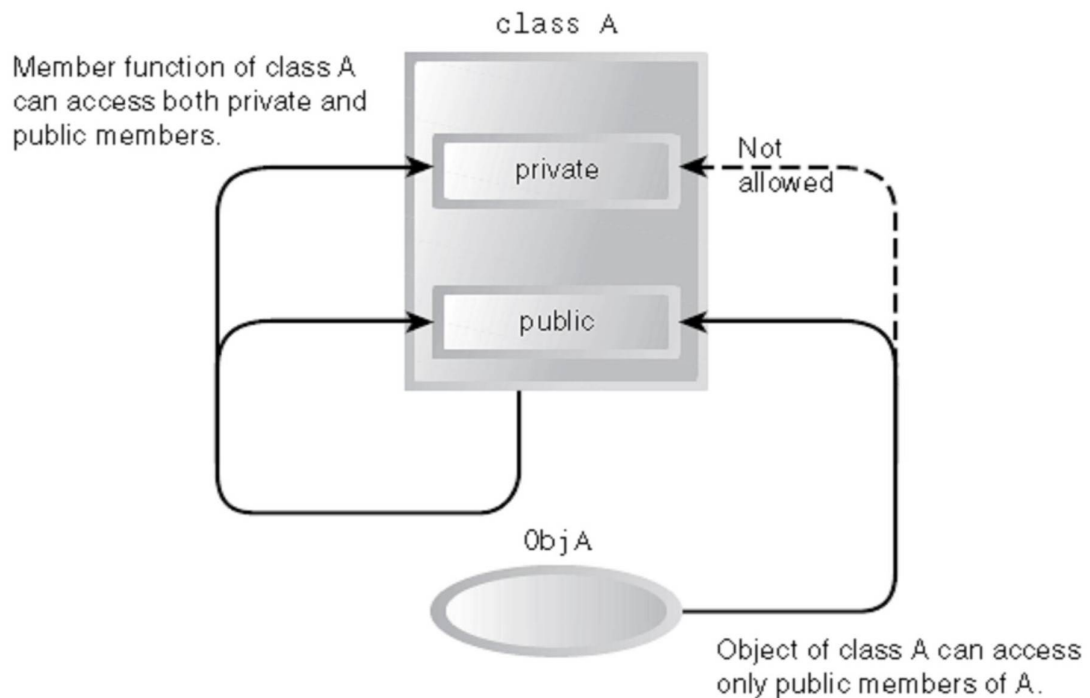
..... // visible to member functions  
 ..... // of its own and derived class

**public:**

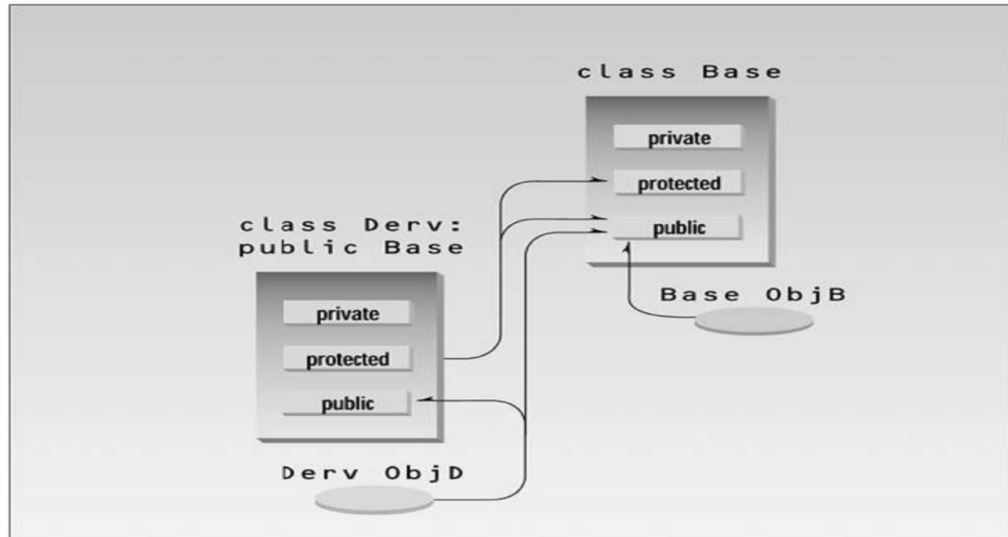
..... // visible to all functions  
 ..... // in the program  
 };

**Table 1: Inheritance and Accessibility**

Access specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside class
<b>Public</b>	yes	yes	yes
<b>Protected</b>	yes	yes	No
<b>private</b>	yes	No	No



**Figure 5.4: Access specifiers without inheritance**



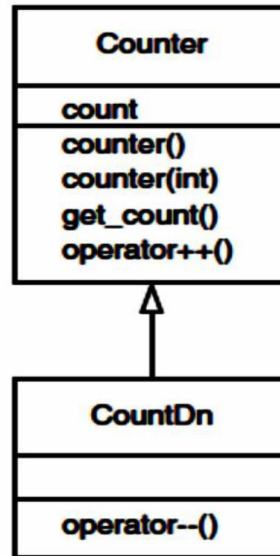
**Figure 5.5: Access specifiers with inheritance**

We can use inheritance to create a new class based on Counter, without modifying counter itself. A new class, CountDn, that adds a decrement operator to the Counter class:

### 5.5 The Unified Modeling Language (UML)

The UML is a graphical “language” for modeling computer programs. “Modeling” means to create a simplified representation of something, as a blueprint models a house. The UML provides a way to visualize the higher-level organization of programs without getting mired down in the details of actual code.

In the UML, inheritance is called generalization, because the parent class is a more general form of the child class. Or to put it another way, the child is more specific version of the parent. The generalization in the COUNTEN program is shown in Figure .5.6. Inheritance



In UML class diagrams, generalization is indicated by a triangular arrowhead on the line connecting the parent and child classes. Remember that the arrow means inherited from or derived from or is a more specific version of. The direction of the arrow emphasizes that the derived class refers to functions and data in the base class, while the base class has no access to the derived class.

Notice that we've added attributes (member data) and operations (member functions) to the classes in the diagram. The top area holds the class title, the middle area holds attributes, and the bottom area is for operations.

**Example:-**

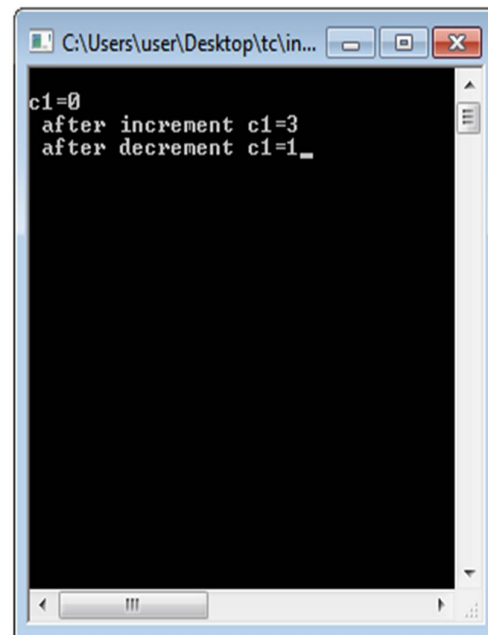
```
#include <iostream>
using namespace std;
class B
{
protected:
    int a; //private; not inheritable
public:
```

```
        int b; //public ready for inheritance
        void get_ab();
        void show_a(void);
};
class D : public B //public derivation
{
    int c;
    public:
        void mul (void);
        void display (void);
};
// function defined //
void B :: get_ab(void)
{
    a = 5;   b = 10;
}
void B :: show_a()
{
    cout << "a = " << a << "\n";
}
void D :: mul ()
{
    c = b * a;
}
void D :: display (void)
{
    cout << " a= " << a << "\n";
    cout << " b= " << b << "\n";
    cout << " c= " << c << "\n";
}
// main program //
main()
{
    D d;
    d.get_ab();
    d.mul();
    d.show_a();
}
```

```
d.display();  
d.b=20;  
d.mul();  
d.display();  
}
```

**Example:- Write a program to decrement the counter variable using inheritance.**

```
#include <iostream >  
using namespace std;  
class Counter  
{  
protected:  
int count; //count  
public:  
Counter (): count (0) //constructor  
{ }  
void inc_count() //increment count  
{ count++; }  
int get_count() //return count  
{ return count; }  
};  
class CountDn : public Counter //derived class  
{  
public:  
void dec_count()  
{ count--; }  
};
```



```
C:\Users\user\Desktop\tc\in...  
c1=0  
after increment c1=3  
after decrement c1=1
```



```
int main()
{
    CountDn c1;           //c1 of class CountDn
    cout << "\nc1=" << c1.get_count(); //display
    c1.inc_count();       //increment c1
    c1.inc_count();       //increment c2
    c1.inc_count();       //increment c2
    cout << "\n after increment c1=" << c1.get_count(); //display again
    c1.dec_count();
    c1.dec_count();
    cout << "\n after decrement c1=" << c1.get_count(); //display again
    return 0;
}
```

### **Output of Example**

In main() we increment c1 three times, print out the resulting value, decrement c1 twice, and finally print out its value again. Here's the output:

c1=0 ← after initialization

c1=3 ← after ++c1, ++c1, ++c1

c1=1 ← after --c1, --c1

The ++ operator, the constructors, the get\_count() function in the Counter class, and the -- operator in the CountDn class all work with objects of type CountDn.

## 5.6 Specifying the Derived Class

Following the Counter class in the listing is the specification for a new class, **CountDn**. This class incorporates a new function, `operator--()`, which decrements the count. However and here's the key point the new CountDn class inherits all the features of the Counter class.

**CountDn** doesn't need a constructor or the `get_count()` or `operator++()` functions, because these already exist in Counter.

The first line of **CountDn** specifies that it is derived from Counter: `class CountDn : public counter`

Here we use a single colon (not the double colon used for the scope resolution operator), followed by the keyword `public` and the name of the base class Counter. This sets up the relationship between the classes. This line says that **CountDn is derived from the base class Counter**.

### Substituting Base Class Constructors

In the main () part of Example we create an object of class CountDn:

```
CountDn c1;
```

This causes `c1` to be created as an object of class CountDn and initialized to 0. But wait how this is possible? There is no constructor in the CountDn class specifier, so what entity carries out the initialization? It turns out that at least under certain circumstances if you don't specify a constructor, the derived class will use an appropriate constructor from the base class.

**In example** there's no constructor in CountDn, so the compiler uses the no argument constructor from Count.

This flexibility on the part of the compiler using one function because another isn't available appears regularly in inheritance situations. Generally, the substitution is what you want, but sometimes it can be unnerving.

### **Substituting Base Class Member Functions**

The object `c1` of the `CountDn` class also uses the `operator++()` and `get_count()` functions from the `Counter` class. The first is used to increment

`c1`:

```
++c1;
```

The second is used to display the count in `c1`:

```
cout << "\nc1=" << c1.get_count();
```

Again the compiler, not finding these functions in the class of which `c1` is a member, uses member functions from the base class.

With inheritance, however, there is a whole raft of additional possibilities. The question that concerns us at the moment is, can member functions of the derived class access members of the base class? In other words, can `operator--()` in `CountDn` access `count` in `Counter`? The answer is that member functions can access members of the base class if the members are public, or if they are protected. They can't access private members.

We don't want to make `count` public, since that would allow it to be accessed by any function anywhere in the program and eliminate the advantages of data hiding. A protected member, on the other hand, can be accessed by member functions in its own class or and here's the key in any

class derived from its own class. It can't be accessed from functions outside these classes, such as main().

## Derived Class Constructors

What happens if we want to initialize a CountDn object to a value? Can the one-argument constructor in Counter be used? The answer is no. As we saw in COUNTEN, the compiler will substitute a no-argument constructor from the base class, but it draws the line at more complex constructors. To make such a definition work we must write a new set of constructors for the derived class. This is shown in the COUNTEN2 program.

**Example :-Write an oop program to decrement the counter variable using constructor in the derived class.**

```
// counten2.cpp

// constructors in derived class

#include <iostream>

using namespace std;

class Counter

{

protected: //NOTE: not private

int count; //count

public:
```

```
Counter() : count() //constructor, no args
```

```
{ }
```

```
Counter(int c) : count(c) //constructor, one arg
```

```
{ }
```

```
int get_count() //return count
```

```
{ return count; }
```

```
Counter operator ++ () //incr count (prefix)
```

```
{ return Counter(++count); }
```

```
};
```

```
class CountDn : public Counter
```

```
{
```

```
public:
```

```
CountDn() : Counter() //constructor, no args
```

```
{ }
```

```
CountDn(int c) : Counter(c) //constructor, 1 arg
```

```
{ }
```

```
CountDn operator -- () //decr count (prefix)
```

```
{ return CountDn(--count); }
```

```
};
```

```
int main()
```

```
{  
  
CountDn c1; //class CountDn  
  
CountDn c2(100);  
  
cout << "\nc1=" << c1.get_count(); //display  
  
cout << "\nc2=" << c2.get_count(); //display  
  
++c1; ++c1; ++c1; //increment c1  
  
cout << "\nc1=" << c1.get_count(); //display it  
  
--c2; --c2; //decrement c2  
  
cout << "\nc2=" << c2.get_count(); //display it  
  
CountDn c3 = --c2; //create c3 from c2  
  
cout << "\nc3=" << c3.get_count(); //display c3  
  
cout << endl;  
  
return 0;  
  
}
```

**Example:-Write an oop program to decrement the counter variable using constructor in the derived class.**

```
// counten2.cpp  
  
// constructors in derived class  
  
#include <iostream>  
  
#include <conio.h>
```

```
using namespace std;

class Counter

{

protected: //NOTE: not private

int count; //count

public:

Counter() : count() //constructor, no args

{ }

Counter(int c) : count(c) //constructor, one arg

{ }

int get_count() //return count

{ return count; }

void inc_count ()

{ ++count; }

};

class CountDn : public Counter

{

public:

CountDn() : Counter() //constructor, no args

{ }
```

```
CountDn(int c) : Counter(c) //constructor, 1 arg
```

```
{ }
```

```
void dec_count() //decr count (prefix)
```

```
{ --count; }
```

```
};
```

```
int main()
```

```
{
```

```
CountDn c1; //class CountDn
```

```
CountDn c2(100);
```

```
cout << "\nc1=" << c1.get_count(); //display
```

```
cout << "\nc2=" << c2.get_count(); //display
```

```
c1.inc_count();
```

```
c1.inc_count();
```

```
c1.inc_count();
```

```
cout << "\nc1=" << c1.get_count(); //display it
```

```
//decrement c2
```

```
c2.dec_count();
```

```
c2.dec_count();
```

```
cout << "\nc2=" << c2.get_count(); //display it
```

```
CountDn c3 = c2; //create c3 from c2
```



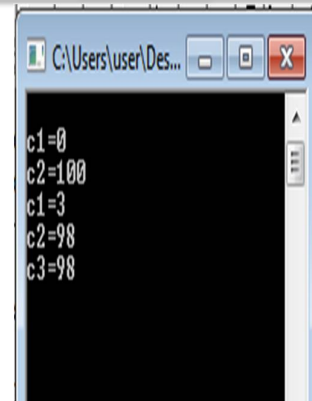
```
cout << "\nc3=" << c3.get_count(); //display c3

cout << endl;

getch();

return 0;

}
```



This program uses two new constructors in the CountDn class. Here is the no-argument constructor:

```
CountDn( ) : Counter( )

{ }
```

This constructor has an unfamiliar feature: the function name following the colon. This construction causes the CountDn( ) constructor to call the Counter( ) constructor in the base class. In main(), when we say

```
CountDn c1;
```

The compiler will create an object of type CountDn and then call the CountDn constructor to initialize it. This constructor will in turn call the Counter constructor, which carries out the work. The CountDn( ) constructor could add additional statements of its own, but in this case it doesn't need to, so the function body between the braces is empty. Calling a constructor from the initialization list may seem odd, but it makes sense. You want to initialize any variables, whether they're in the derived class or the base class, before any statements in either the derived or base-class constructors are executed. By calling the baseclass constructor before the derived-class constructor starts to execute, we accomplish this. The statement

```
CountDn c2(100);
```

In main() uses the one-argument constructor in CountDn. This constructor also calls the corresponding one-argument constructor in the base class:

```
CountDn(int c) : Counter(c) argument c is passed to Counter
```

```
{ }
```

This construction causes the argument c to be passed from CountDn() to Counter(), where it is used to initialize the object. In main(), after initializing the c1 and c2 objects, we increment one and decrement the other and then print the results. The one-argument constructor is also used in an assignment statement.

```
CountDn c3 = --c2;
```

## 5.7 Overriding Member Functions

You can use member functions in a derived class that override that is, have the same name as those in the base class. You might want to do this so that calls in your program work the same way for objects of both base and derived classes. The program modeled a stack, a simple data storage device. It allowed you to push integers onto the stack and pop them off. However, STAKARAY had a potential flaw. If you tried to push too many items onto the stack, the program might bomb, since data would be placed in memory beyond the end of the st[ ] array. Or if you tried to pop too many items, the results would be meaningless, since you would be reading data from memory locations outside the array. To cure these defects we've created a new class, Stack2, derived from Stack. Objects of Stack2 behave in exactly the same way as those of Stack, except that you will be warned if you

attempt to push too many items on the stack or if you try to pop an item from an empty stack. Here's the listing for STAKEN:

**Example:- Write an oop program to overload functions in base and derived stack classes.**

```
// staken.cpp

#include <iostream.h>

#include <process.h> //for exit()

class Stack

{

protected: //NOTE: can't be private

enum { MAX = 3 }; //size of stack array

int st[MAX]; //stack: array of integers

int top; //index to top of stack

public:

Stack() //constructor

{ top = -1; }

void push(int var) //put number on stack

{ st[++top] = var; }

int pop() //take number off stack

{ return st[top--]; }

};
```

```
class Stack2 : public Stack

{

public:

void push(int var) //put number on stack

{

if(top >= MAX-1) //error if stack full

{ cout << "\nError: stack is full"; exit(1); }

Stack::push(var); //call push() in Stack class

}

int pop() //take number off stack

{

if (top < 0) //error if stack empty

{

cout << "\nError: stack is empty\n"; exit(1); }

return Stack::pop(); //call pop() in Stack class

}

};

int main()

{

Stack2 s1;
```

```
s1.push(11); //push some values onto stack

s1.push(22);

s1.push(33);

cout << endl << s1.pop(); //pop some values from stack

cout << endl << s1.pop();

cout << endl << s1.pop();

cout << endl << s1.pop(); //oops, popped one too many...

cout << endl;

return 0;

}
```

In this program the Stack class is just the same as it was in the STAKARAY program, except that the data members have been made protected.

### **Which Function Is Used?**

The Stack2 class contains two functions, push() and pop(). These functions have the same names and the same argument and return types, as the functions in Stack. When we call these functions from main(), in statements like

```
s1.push(11);
```

How does the compiler know which of the two push() functions to use? Here's the rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed.

(This is true of objects of the derived class. Objects of the base class don't know anything about the derived class and will always use the base class functions.)

We say that the derived class function overrides the base class function. So in the preceding statement, since s1 is an object of class Stack2, the push() function in Stack2 will be executed, not the one in Stack. The push() function in Stack2 checks to see whether the stack is full. If it is, it displays an error message and causes the program to exit. If it isn't, it calls the push() function in Stack. Similarly, the pop() function in Stack2 checks to see whether the stack is empty. If it is, it prints an error message and exits; otherwise, it calls the pop() function in Stack. In main() we push three items onto the stack, but we pop four. The last pop elicits an error message

33

22

11

Error: stack is empty

and terminates the program.

### **5.7.1 Scope Resolution with Overridden Functions**

How do push() and pop() in Stack2 access push() and pop() in Stack? They use the scope resolution operator, ::, in the statements

```
Stack::push(var);
```

and

```
return Stack::pop();
```

These statements specify that the push() and pop() functions in Stack are to be called. Without the scope resolution operator, the compiler would think the push() and pop() functions in Stack2 were calling themselves, which in this case would lead to program failure. Using the scope resolution operator allows you to specify exactly what class the function is a member of.

## 5.8 Inheritance in the English Distance Class

Here's a somewhat more complex example of inheritance. So far in this book the various programs that used the English Distance class assumed that the distances to be represented would always be positive. This is usually the case in architectural drawings. However, if we were measuring, say, the water level of the Pacific Ocean as the tides varied, we might want to be able to represent negative feet-and-inches quantities. (Tide levels below mean-lower-low-water are called minus tides; they prompt clam diggers to take advantage of the larger area of exposed beach.) Let's derive a new class from Distance. This class will add a single data item to our feet-and inches measurements: a sign, which can be positive or negative. When we add the sign, we'll also need to modify the member functions so they can work with signed distances. Here's the listing for ENGLLEN:

**Example:-Write an oop program to overload functions in base and derived distance classes**

```
// englen.cpp  
  
// inheritance using English Distances  
  
#include <iostream>
```

```
using namespace std;

enum posneg { pos, neg }; //for sign in DistSign

class Distance //English Distance class

{

protected: //NOTE: can't be private

int feet;

float inches;

public: //no-arg constructor

Distance() : feet(0), inches(0.0)

{ } //2-arg constructor

Distance(int ft, float in) : feet(ft), inches(in)

{ }

void getdist() //get length from user

{

cout << "\nEnter feet: "; cin >> feet;

cout << "Enter inches: "; cin >> inches;

}

void showdist() const //display distance

{ cout << feet << "\'-" << inches << "\'"; }

};

class DistSign : public Distance //adds sign to Distance
```



```
{  
  
private:  
  
posneg sign; //sign is pos or neg  
  
public:  
  
//no-arg constructor  
  
DistSign() : Distance() //call base constructor  
  
{ sign = pos; } //set the sign to +  
  
//2- or 3-arg constructor  
  
DistSign(int ft, float in, posneg sg=pos) :  
Distance(ft, in) //call base constructor  
  
{ sign = sg; } //set the sign  
  
void getdist() //get length from user  
  
{  
  
Distance::getdist(); //call base getdist()  
  
char ch; //get sign from user  
  
cout << "Enter sign (+ or -): "; cin >> ch;  
  
sign = (ch=='+' ? pos : neg;  
  
}  
  
void showdist() const //display distance  
  
{
```

```
cout << ( (sign==pos) ? “(+)” : “(-)” );           //show sign

Distance::showdist(); //ft and in

}

};

int main()

{

DistSign alpha;                               //no-arg constructor

alpha.getdist();                               //get alpha from user

DistSign beta(11, 6.25);                       //2-arg constructor

DistSign gamma(100, 5.5, neg);                 //3-arg constructor

//display all distances

cout << “\nalpha = “; alpha.showdist();

cout << “\nbeta = “; beta.showdist();

cout << “\ngamma = “; gamma.showdist();

cout << endl;

return 0;

}
```

Here the DistSign class adds the functionality to deal with signed numbers. The Distance class in this program is just the same as in previous programs, except that the data is protected. Actually in this case it could be private, because none of the derived-class functions accesses it. However,

it's safer to make it protected so that a derived-class function could access it if necessary.

### **Operation of ENGLLEN**

The main() program declares three different signed distances. It gets a value for alpha from the user and initializes beta to (+)11'-6.25" and gamma to (-)100'-5.5". In the output we use parentheses around the sign to avoid confusion with the hyphen separating feet and inches. Here's some sample output:

Enter feet: 6

Enter inches: 2.5

Enter sign (+ or -): -

alpha = (-)6'-2.5"

beta = (+)11'-6.25"

gamma = (-)100'-5.5"

The DistSign class is derived from Distance. It adds a single variable, sign, which is of type posneg. The sign variable will hold the sign of the distance. The posneg type is defined in an enum statement to have two possible values: pos and neg.

### **Constructors in DistSign**

DistSign has two constructors, mirroring those in Distance. The first takes no arguments, the second takes either two or three arguments. The third, optional, argument in the second constructor is a sign, either pos or neg. Its default value is pos. These constructors allow us to define variables (objects) of type DistSign in several ways. Both constructors in DistSign

call the corresponding constructors in Distance to set the feet-and- inches values. They then set the sign variable. The no-argument constructor always sets it to pos. The second constructor sets it to pos if no third-argument value has been provided, or to a value (pos or neg) if the argument is specified. The arguments ft and in, passed from main() to the second constructor in DistSign, are simply forwarded to the constructor in Distance.

### **Member Functions in DistSign**

Adding a sign to Distance has consequences for both of its member functions. The getdist() function in the DistSign class must ask the user for the sign as well as for feet-and-inches values, and the showdist() function must display the sign along with the feet and inches. These functions call the corresponding functions in Distance, in the lines

```
Distance::getdist();
```

and

```
Distance::showdist();
```

These calls get and display the feet and inches values. The body of getdist() and showdist() in DistSign then go on to deal with the sign. Lecture

### **Example:-**

1. Imagine a publishing company that markets both book and audiocassette versions of its works. Create a class publication that stores the title (a int) and price (type float) of a publication. From this class derive two classes: book, which adds a page count (type int), and tape, which adds a playing time in minutes (type float). Each of these three classes should have a

getdata() function to get its data from the user at the keyboard, and a putdata() function to display its data. Write a main() program to test the book and tape classes by creating instances of them, asking the user to fill in data with getdata(), and then displaying the data with putdata().

2. Start with the publication, book, and tape classes of Example 1. Add a base class sale that holds an array of three floats so that it can record the dollar sales of a particular publication for the last three months. Include a getdata() function to get three sales amounts from the user, and a putdata() function to display the sales figures. Alter the book and tape classes so they are derived from both publication and sales. An object of class book or tape should input and output sales data along with its other data. Write a main() function to create a book object and a tape object and exercise their input/output capabilities.

### **Solutions to Example**

1.

```
// ex1.cpp
```

```
// publication class and derived classes
```

```
#include <iostream >
```

```
#include <string.h>
```

```
using namespace std;
```

```
class publication // base class
```

```
{
```

```
private:
```

```
int title;
```

```
float price;

public:

void getdata()

{

cout << "\nEnter title: "; cin >> title;

cout << "Enter price: "; cin >> price;

}

void putdata() const

{

cout << "\nTitle: " << title;

cout << "\nPrice: " << price;

}

};

class book : private publication // derived class

{

private:

int pages;

public:

void getdata()

{
```

```
publication::getdata();

cout << "Enter number of pages: "; cin >> pages;
}

void putdata() const
{
publication::putdata();

cout << "\nPages: " << pages;

}

};

class tape : private publication // derived class
{
private:

float time;

public:

void getdata()
{
publication::getdata();

cout << "Enter playing time: "; cin >> time;

}

void putdata() const
{
```

```
publication::putdata();

cout << "\nPlaying time: " << time;

}

};

int main()

{

book book1; // define publications

tape tape1;

book1.getdata(); // get data for them

tape1.getdata();

book1.putdata(); // display their data

tape1.putdata();

cout << endl;

return 0;

}
```

**2.**

```
// ex2.cpp

// multiple inheritance with publication class

#include <iostream>

#include <string.h>
```



```
using namespace std;

class publication

{ private:

int title;

float price;

public:

void getdata()
{

cout << "\nEnter title: "; cin >> title; cout << " Enter price: "; cin >> price;
}

void putdata() const

{

cout << "\nTitle: " << title; cout << "\n Price: " << price; }

};

class sales

{ private:

enum { MONTHS = 3 };

float salesArr[MONTHS];

public:

void getdata();

void putdata() const;
```

```
};

void sales::getdata()

{

cout << " Enter sales for 3 months\n"; for(int j=0; j<MONTHS; j++) {

cout << " Month " << j+1 << ": "; cin >> salesArr[j]; } }

void sales::putdata() const

{

for(int j=0; j<MONTHS; j++)

{ cout << "\n Sales for month " << j+1 << ": "; cout << salesArr[j]; } }

class book : private publication, private sales

{

private:

int pages;

public:

void getdata()

{ publication::getdata();

cout << " Enter number of pages: "; cin >> pages;

}

void putdata() const

{

publication::putdata();
```

```
cout << "\n Pages: " << pages;

sales::putdata();

}

};

class tape : private publication, private sales
{

private:

float time;

public:

void getdata()

{

publication::getdata();

cout << " Enter playing time: "; cin >> time;

sales::getdata();

}

void putdata() const

{

publication::putdata();

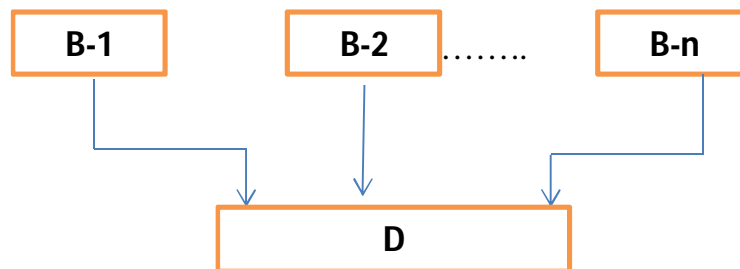
cout << "\n Playing time: " << time;

sales::putdata();
```

```
} };  
  
int main()  
{  
  
book book1;           // define publications  
  
tape tape1;  
  
book1.getdata();     // get data for publications  
  
tape1.getdata();  
  
book1.putdata();     // display data for publications  
  
tape1.putdata();  
  
cout << endl;  
  
return 0;
```

### 5.9 Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in figure 5.7. This 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 and the intelligence of another.



**Figure 5.7 Multiple inheritance**

The syntax of a derived class with multiple base classes is as follows:

```
Class D: visibility B-1, visibility B-2 .....  
{  
.....  
..... (Body of D)  
.....  
};
```

Where, visibility may be either **public** or **private**. The base classes are separated by commas.

**Example:-**

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

classes M and N have been specified as follows:

```
class M  
  
{  
  
protected:  
  
int m;
```

```
public:

void get_m(int);

};

void M:: get_m(int x)

{

m=x;

}

class N

{

protected:

int n;

public:

void get_n(int);

};

void N:: get_n(int x)

{

n=y;

}
```

The derived class **P**, as declared above, would, in effect, contain all the members of **M** and **N** in addition to its own members as shown below:

```
class p
```

```
{  
  
protected:  
  
    Int m;                // from M  
    Int n;                // from N  
  
Public:  
  
    Void get_m(int);     // from M  
    Void get_n(int);     // from N  
    Void display(void);  // Own member  
  
};
```

The member function display() can be defined as follows:

```
Void p::display(void)  
  
{  
  
    cout<<"m="<<m<<"\n";  
  
    cout<<"n="<<n<<"\n";  
  
    cout<<"m*n="<<m*n<<"\n";  
  
};
```

The main() function which provides the user-interface may be written as follows:

```
main()  
  
{  
  
Pp;
```

```
P.get_m(10);
```

```
P.get_n(20);
```

```
P.display();
```

```
}
```

Program shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.

```
// multiple inheritance //
```

```
#include<iostream>
```

```
using namespace std;
```

```
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);

};

void M:: get_m(int x)

{

m=x;

}

void N:: get_n(int y)

{

n=y;

}

void P:: display(void)

{

cout<<"m="<<m<<"\n";

cout<<"n="<<n<<"\n";

cout<<"m*n="<<m*n<<"\n";
```

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

**The output of program would be:**

m=10

n=20

m\*n=200

### **Ambiguity Resolution in Inheritance**

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class.

Consider the following two classes.

```
class M  
  
{  
  
public:
```

```
void display(void)
```

```
{
```

```
cout<<"class M\n";
```

```
}
```

```
};
```

```
class N
```

```
{
```

```
public:
```

```
void display(void)
```

```
{
```

```
cout<<"class N\n";
```

```
}
```

```
};
```

Which **display()** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as show below:

```
class P: public M, public N
```

### 5.10 Multi-level inheritance

It is not uncommon that a class is derived from another derived class as shown in fig. 8.6. The 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. The 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.

```
class student
{
    protected:
        int roll-number;
    public:
        void get_ number (int);
        void put_ number (void);
};
void student :: get_ number (int a)
{
    roll_ number = a;
}
void student :: put_ number ()
{
    cout <<"Roll Number:" <<roll_ number <<"\n";
}
class test: public student    // first level derivation
protected:
float su1;
float sub2;
public:
void get_marks(float, float);
```

```
void put_marks(void);

};

void test::get_marks(float x, float y)

{
sub1=x;
sub2=y;
}

void test::put_marks()

{
cout<<"marks in sub1="<<sub1<<"\n";
cout<<"marks in sub2="<<sub2<<"\n";
}

class result:public test    // second level derivation
{
float total;                // private by default
public:
void display(void);
};

private:
float total;                // own member
protected:
int roll_number;           // inherited from student via test
float sub1;                 // inherited from test
float sub2;                 // inherited from test
public:
void get_number(int);       // from student via test
void put_number(void);      // from student via test
void get_marks(float, float); // from test
void put_marks(void);       // from test
```

```
void display(void);          // own member
```

The inherited functions **put\_number()** and **put\_marks()** can be used in the definition of **display()** function:

```
void result::display(void)
{
total = sub1+sub2;
put_number();
put_marks();
cout<<"total="<<total<<"\n";
}
```

Here is a simple main() program:

```
int main()
{
result student1;          // student created
student1.get_number(111);
student1.get_marks(75,0,59,5);
student1.display();
return 0;
}
```

This will display the result of **student1**. The complete program is shown in program:

```
//multilevel inheritance //
#include<iostream>
using namespace std;
class student
{
protected:
```

```
int roll-number;

public:
    void get_ number (int);
    void put_ number (void);
};

void student :: get_ number (int a)
{
    roll_ number = a;
}

void student :: put_ number ()
{
    cout <<"Roll Number:" <<roll_ number <<"\n";
}

class test: public student    // first level derivation
{
protected:
float sub1;
float sub2;
public:
void get_marks(float, float);
void put_marks(void);
};

void test:: get_marks(float x, float y)
{
sub1=x;
sub2=y;
}

void test::put_marks()
{
```

```
cout<<"marks in sub1="<<sub1<<"\n";
cout<<"marks in sub2="<<sub2<<"\n";
}
class result:public test    // second level derivation
{
float total;                // private by default
public:
void display(void);
};
void result:: display(void)
{
total = sub1+sub2;
put_number();
put_marks();
cout<<"total="<<total<<"\n";
}
int main()
{
result student1;           // student created
student1.get_number(111);
student1.get_marks(75,0,59,5);
student1.display();
return 0;
}
```

**Program display the following output:**

Roll number 11

Marks in sub1=75

Marks in sub1=59.5

Total=134.5