الفصل الخامس

الوراثـــــــــه

Inheritance

WPS Office تعديل من خلال

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 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.

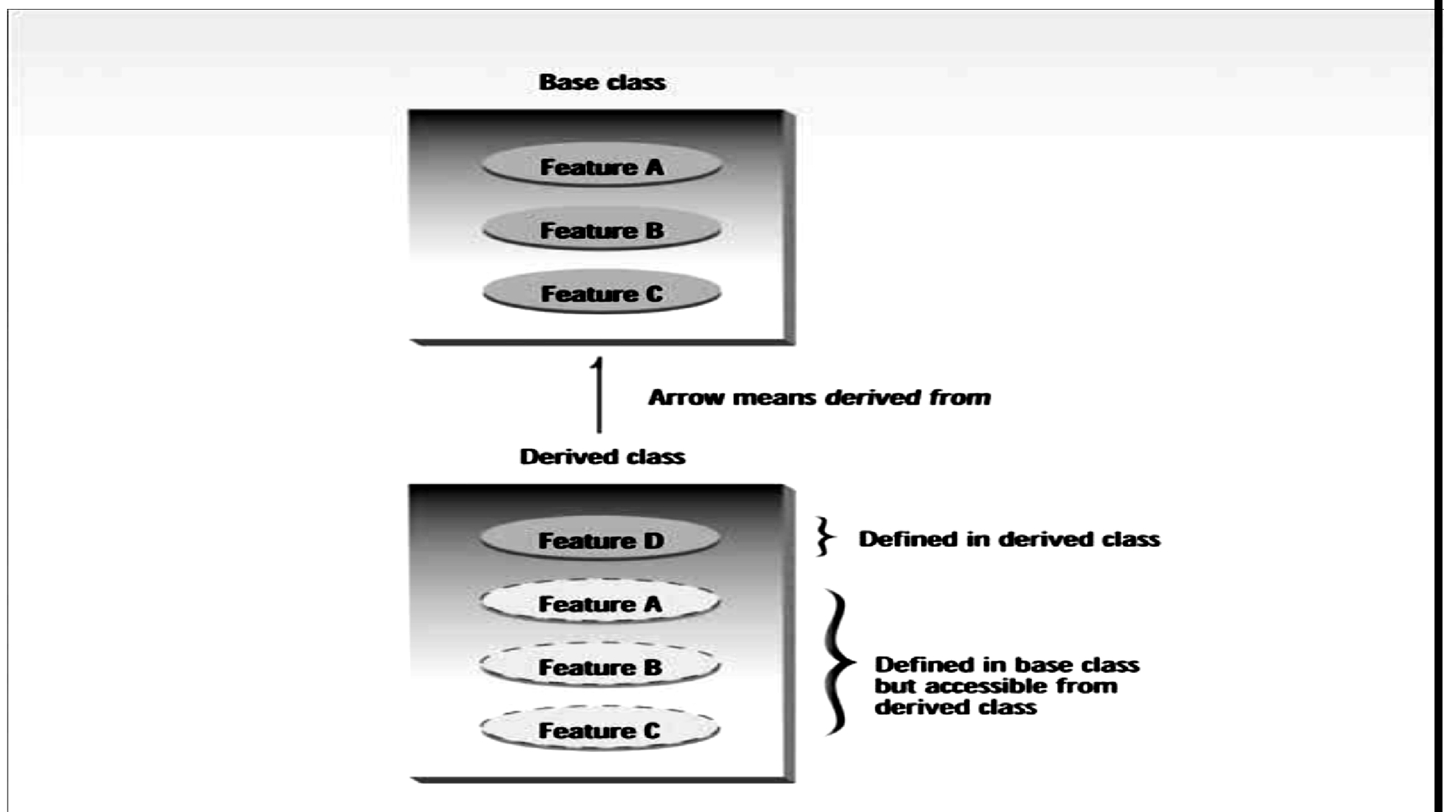


FIGURE (5.1) *Inheritance.*

Inheritance is an essential part of OOP. Its code reusability, Once a base class is written and debugged, it need not be touched again, but, using inheritance, can nevertheless be adapted to work in different situations.
Reusing existing code saves time and money and increases a program's reliability. Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.. An important result of reusability is the ease of distributing class libraries.
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. We'll

WPS Office تعديل من خلال

examine these features of inheritance in more detail after we've seen some specific instances of inheritance at work.

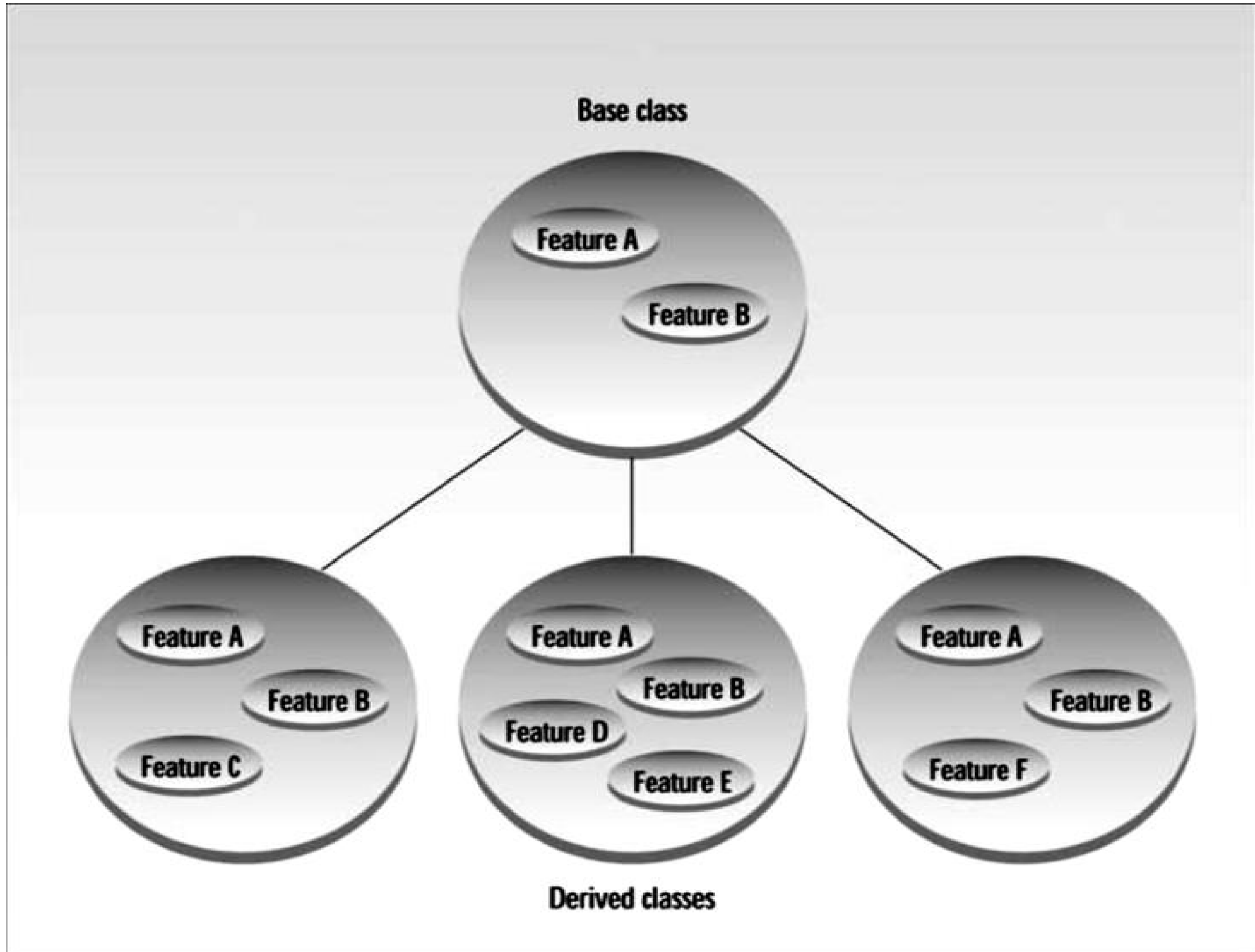


Fig 5.2

Reusability is another important feature of oop. C++ strongly supports the concept of reusability. 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 a 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*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level.
- *single inheritance* :- a derived class with only one base class
- *multiple inheritance* :-one derived class with several base classes.

WPS Office تعديل من خلال

- *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'.

Figure 5.3 shows various form of inheritance. The direction of arrow indicates the direction of inheritance.

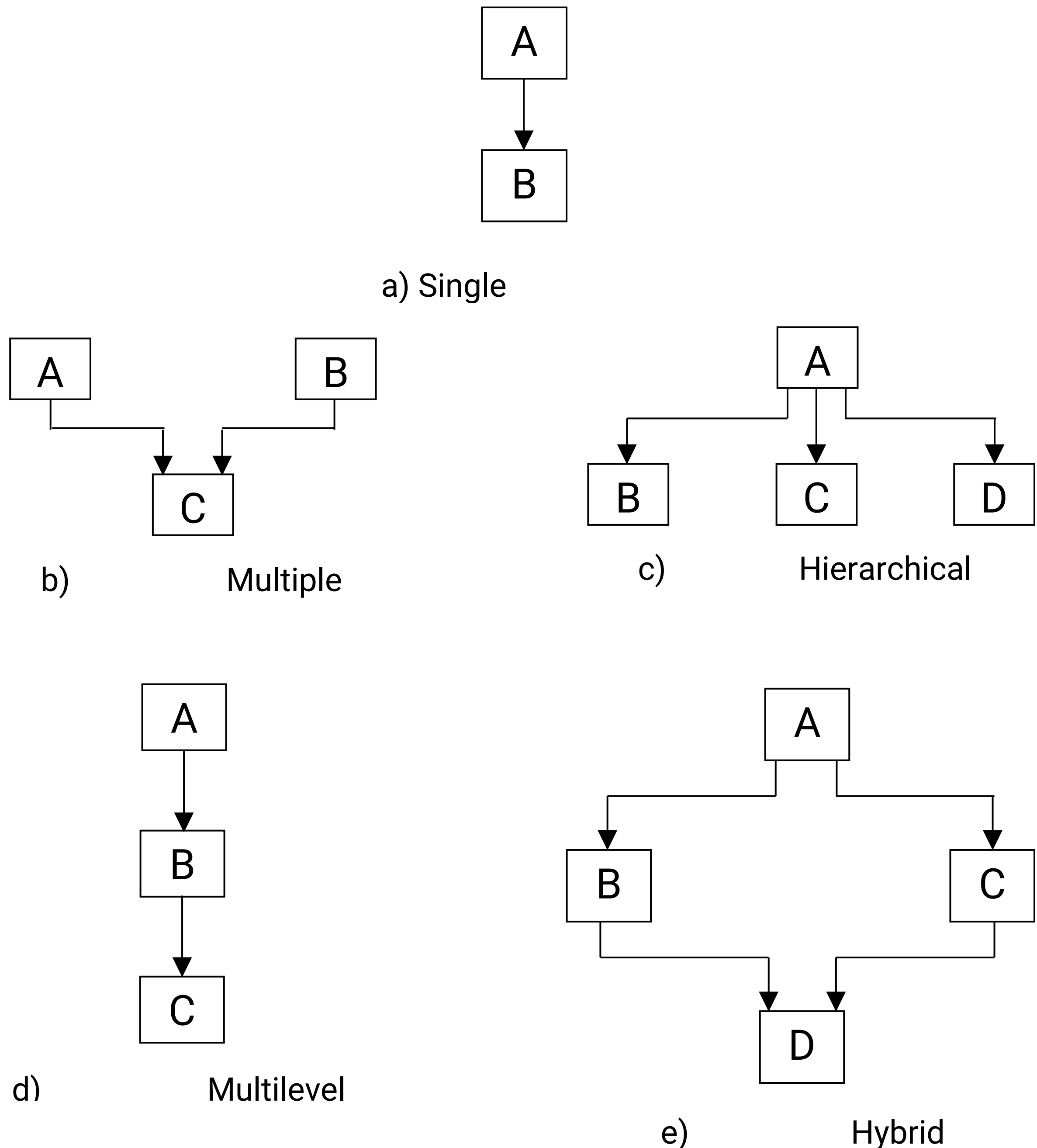


Fig (5.3) Forms of inheritance

WPS Office تعديل من خلال

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.*
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
};
```

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

WPS Office تعديل من خلال

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 functionsr.

```
////////////////////////////////////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();
```

The output of program:
a=5
a=5
b=10
c=50

a=5
b=20
c=100

WPS Office تعديل من خلال

```
   d.mul();
   d.show_a();
   d.display();
   d.b=20;
   d.mul();
   d.display();
return 0;
}
```

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 fig 5.4.

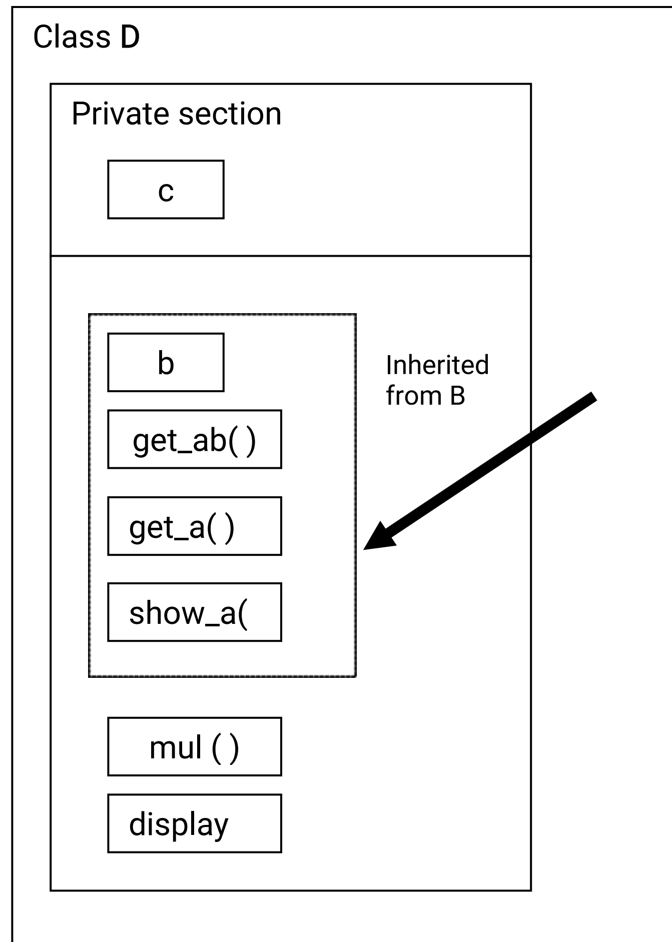


Fig 5.4 Adding more members to a class

WPS Office تعديل من خلال

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 as 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. 5.5.

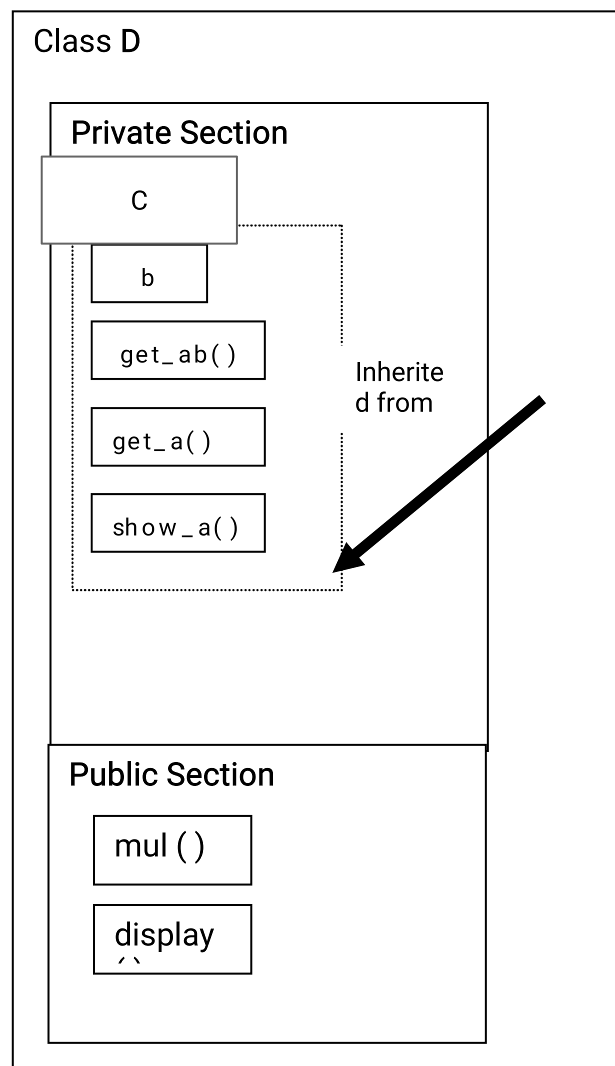


Fig 5.5 Adding more members to a class

In **private** derivation the public mem[illegible] (by private derivation) [illegible]e members of the derived class. Therefore, the objects of D cannot have direct access to the public member functions of B.

WPS Office تعديل من خلال

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 ( );
}
```

```
/////////////////////////// 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)
```


WPS Office تعديل من خلال

```
{
    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
  ………               // visibile to member functions
   ……..               // within its class
  protected :
   …….             // visibile to member functions
   ……..           // of its own and derived class
  public:
  ……..          // visible to all functions
   …….          // in the program
   };
```

Table 1: Inheritance and Accessibility

WPS Office تعديل من خلال

Access Specifier	Accessible from Own Class	Accessible from Derived Class	Accessible from Objects Outside Class
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

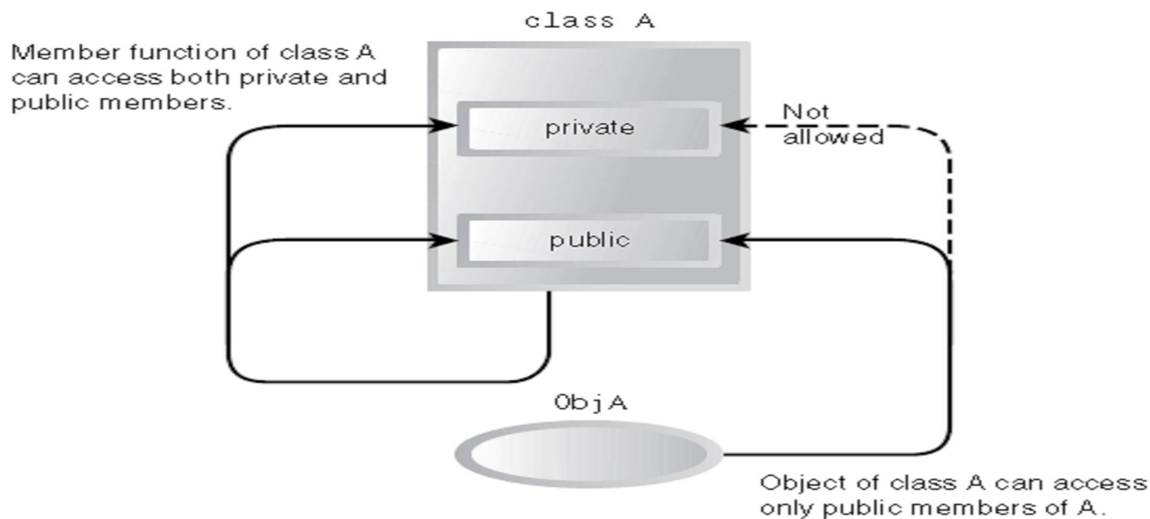


Figure 1 Figure 5.6: Access *specifiers without inheritance*

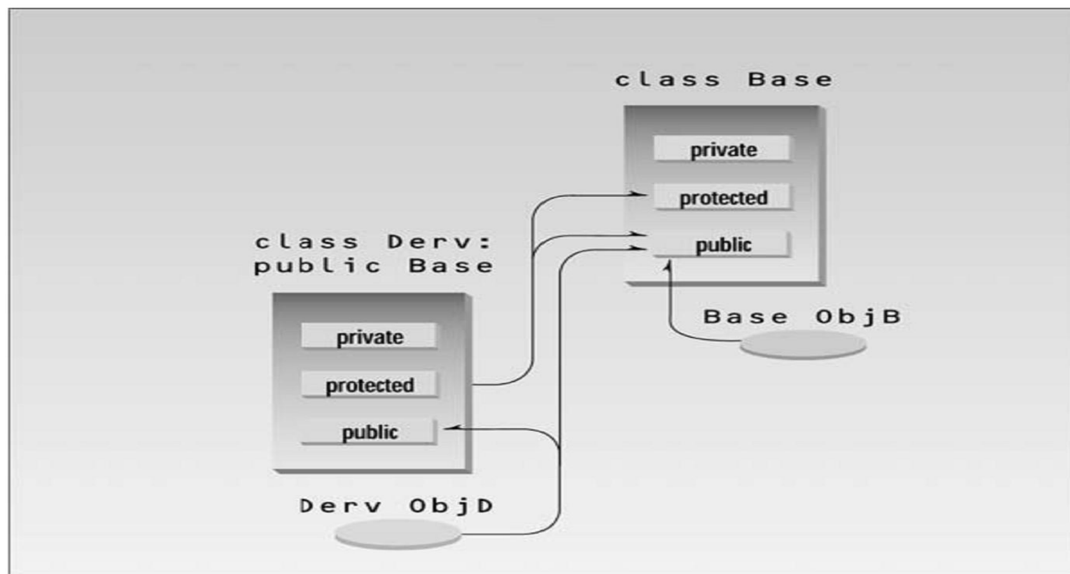


Figure 2 Figure 7.7: *Access specifiers with inheritance*

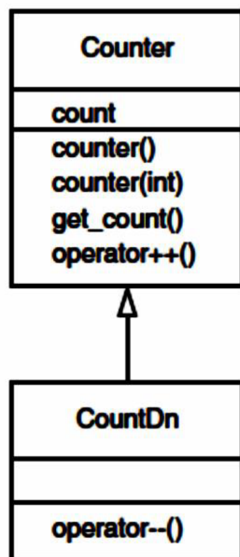
we can use inheritance to create a new class based on Counter, without modifying

WPS Office تعديل من خلال

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 .3. 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.

Ex: W.P define class ….

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


WPS Office تعديل من خلال

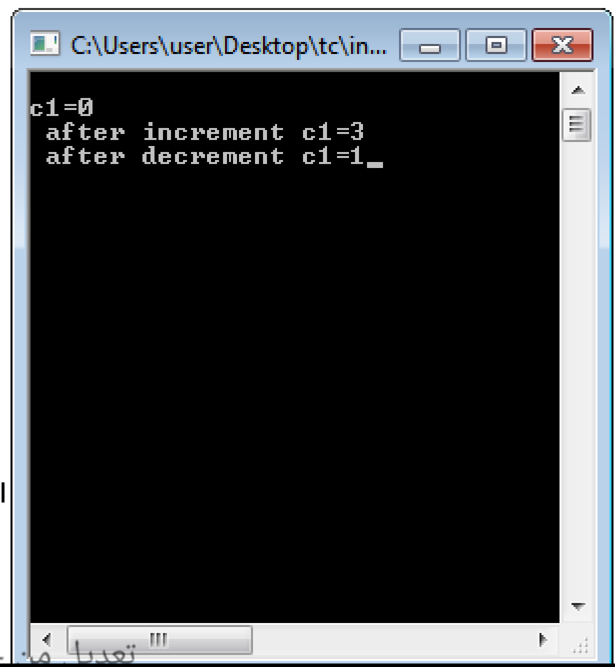
```
    public:
          int b;    //public ready for inheritance
          void  get_ab( );
          void  show_a(void);
};
class D : public B   //public derivation
{      int c;
    public:
          void  mul (void);
          void  display (void);
};
// ……………………… function defined …………………..
void 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……………………….
int main()
{    D d;

    d.get_ab();
    d.mul();
    d.show_a();
    d.display();

    d.b=20;
    d.mul();
    d.display();
return 0;
}
```

EX: Write a program to decrement the counter




```
#include <iostream>
```


WPS Office تعديل من خلال

```
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--; }
};

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.

WPS Office تعديل من خلال

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 Example1 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 is this 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 **example1** 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.

5.7 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

WPS Office تعديل من خلال

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().

5.7.1 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)
```

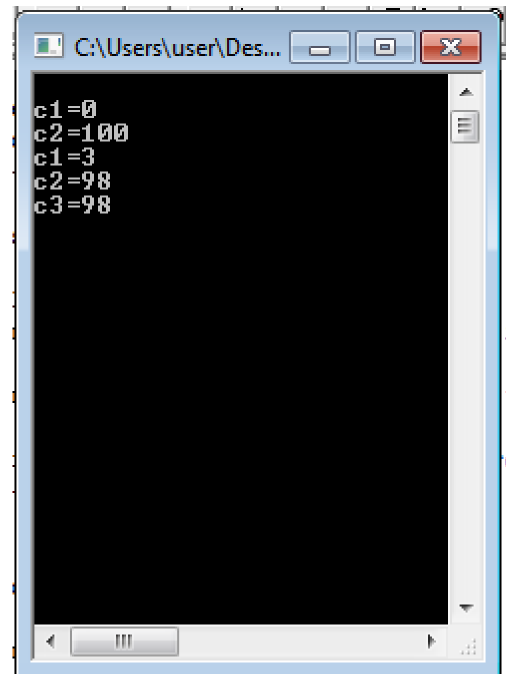

WPS Office تعديل من خلال

```
{ 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
```



WPS Office تعديل من خلال

```
{ }
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
 c2.dec_count();                                  //decrement c2
 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

WPS Office تعديل من خلال

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;

WPS Office تعديل من خلال