Binding, Polymorphisms, and Virtual Functions

CHAPTER 15

CHAPTER OUTLINE

15.1 INTRODUCTION

One of the key features offered by C++ is polymorphisms. It is essential to know the concepts of polymorphisms and associated topics. In this chapter, we are going to learn and implement static (early) binding, dynamic (late) binding, polymorphisms, and virtual functions. An attempt is made to illustrate every point of this new topic in an easy way, and complicated subtopics are explained in a simple way.

The word `poly` means `many`, and `morphism` means `several` forms. Both the words are derived from Greek language. Thus, by combining these two words, a new whole word called

polymorphism is created, which means various forms. We have learnt about overloading of functions and operators. It is also one type of polymorphism. The information pertaining to various overloaded member functions and arguments is known to the compiler while compiling. Thus, the compiler selects an appropriate function and collects the arguments at compile time. This is called early binding or static binding. This is also known as compile time polymorphism.

In C++, the function can be bound at either compile time or run time. Deciding a function call at compile time is called compile time or early or static binding. Deciding a function call at run time is called run time or late or dynamic binding. Dynamic binding permits to suspend the decision of choosing a suitable member function until run time. Two types of polymorphism are shown in Figure 15.1.

POLYMORPHISMS

A polymorphism is a technique in which various forms of a single function can be defined and shared by various objects to perform an operation.

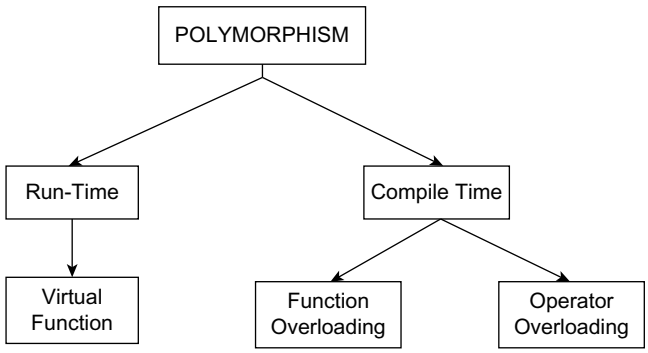

Fig. 15.1 Polymorphisms in C++

15.2 BINDING IN C++

Though C++ is an object-oriented programming language, it is very much inspired by procedural language. A program in C++ is executed sequentially line by line. Each line of the source program after translation is in the form of machine language. Each line in the machine language is assigned a unique address. Similarly, when a function call is encountered by the complier during execution, the function call is translated into machine language, and a sequential address is provided. Thus, binding refers to the process that is to be used for converting functions and variables into machine language addresses. The C++ supports two types of binding: static or early binding and dynamic or late binding.

15.2.1 Static (Early) Binding

In this section, we are going to focus on static or early binding. Even though similar function names are used at many places, their references and their positions are indicated explicitly by the compiler. Their ambiguities are fixed at compile time. Consider the following example.

```
class first // base class
{
   int d;
   public:
   void display() {------} // base  class  member
                              function
};
class second : public first // derived class
{
   int k:
   public:
   void display() {-------} // member function of
derived class
}
```

In the above program, base and derived classes have a similar member function name `display()` but after conversion into machine language, addresses for two `display()` functions are different and unique. Thus, the compiler instructs the CPU to jump to the different addresses when similar `display()` function calls are executed at various places in the program. This is not an overloaded function, because its prototype is the same and it is defined in different classes.

The following program is based on early binding:

15.1 Write a program to invoke function using scope access operator.

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

class first
{
   int b;
   public:
   first() {b=10;}
   void display()   {cout<<"\n b="<<b;}
};
class second: public first
{
   int d;
   public:
   second() {d=20;}
   void display()   {cout<<"\n d="<<d;}
};
 int main()
  {
   clrscr();
```

```
    second s;
    s.first::display();    // Invokes base class function
    s.display();          // Invokes derived class function
    return 0;
  }
```

OUTPUT

b = 10
d = 20

Explanation: In the above program, the class first is a base class and class second is a derived class. Both the classes contain one integer data member and member function. The `display()` function is used to display the content of the data member. Both the classes contain a similar function name. In function `main()`, `s` is an object of the derived `class second`. Hence, in order to invoke the `display()` function of the base class, the scope access operator is used. When base and derived classes have similar function names, in such a situation, it is very essential to provide information to the compiler at run time about the member functions. The mechanism that provides run-time selection of a function is called a polymorphism.

A `virtual` keyword plays an important role in late binding. Before introducing virtual functions, let us have a look at the given below program. The following program shows what happens when functions are not declared `virtual`. This is an example of early binding.

15.2 Write a program to invoke member function of base and derived class using pointer of base class.

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

class first
{  int b;
   public:
   first() {b=10;}
   void display()   {cout<<"\n b=" <<b;}
};
class second: public first
{
   int d;
   public:
   second() {d=20;}
   void display()  {cout<<"\n d="<<d;}
};
int main()
  {
   clrscr();
   first f,*p;
```

```
    second s;
    p=&f;
    p->display();
    p=&s;
    p->display();
    return 0;
  }
```

OUTPUT

b = 10
b = 10

Explanation: In the above program, the `class first` is a base class and `class second` is a derived class. The variable `f` is an object of the base class, and `s` is an object of the derived class. The pointer `*p` is an object pointer of the base class. The address of the base class object is assigned to pointer `p`**,** and `display()` function is called. Again, the address of the derived class is assigned to pointer `p`**,** and `display()` function is called. Both the times, the `display()` function of base class is executed. This is so, because though in the second call `p` contains the address of the object of the derived class, yet the `display()` function of the base class replaces the existence of the `display()` function of the derived class. In order to execute various forms of the same function defined in the base, derived class, run-time binding is necessary, and it can be achieved using the `virtual` keyword.

15.2.2 Dynamic (Late) Binding

In the case of a few programs, it is impossible to know which function is to be called until run time. This is called `dynamic binding`. Dynamic binding of member functions in C++ can be done using the `virtual` keyword. The member function followed by the `virtual` keyword is called a `virtual` function. Consider the following example:

```
class first // base class
{
   int d;
   public:
   virtual void display() {------} // base class member
function
};
class second: public first // derived class
{
   int k:
   public:
   virtual void display() {-------} // member function
of derived class
}
```

In the example of the above program, base and derived classes have a similar member function `display()` preceded by the keyword `virtual`. The various forms of virtual functions in base and derived classes are dynamically bound. The references are detected from the base class. All virtual functions in derived classes are considered virtual, supposing they match the base class function exactly in the number and types of parameters sent. If there is no match between these functions, they will be assumed as overloaded functions. The virtual function should be defined in the public section. If the function is declared virtual, the system will use dynamic (late) binding, which is carried out at run time. Otherwise, early or compile-time binding is used.

Dynamic binding can be implemented with function pointers. In this method, the pointer points to a function instead of a variable. A simple programming example is given below based on late binding.

15.3 Write a program to perform few arithmetic operations on floating numbers using functions. Use function pointer (Late binding).


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

float add(float m, float n)
{
   return m+n;
}
float sub(float m, float n)
{
   return m-n;
}
float mul(float m, float n)
{
   return m*n;
}
int main()
{
   clrscr();
   float x,y;
   cout<<"Enter two numbers:";
   cin>>x>>y;
   int task;
   do
   {
   cout<<"Enter task (1=add, 2=sub, 3=mul):";
   cin>>task;
   }
   while (task< 1 || task > 3);
```

```
    float (*pt)(float, float);
    // Function pointer named pt is created
    switch (task)
    {
    case 1:
    pt=add;
    break;
    case 2:
    pt=sub;
    break;
    case 3:
    pt=mul;
    break;
    }
    cout<<"Result of operation is:" << pt(x,y) << endl;
    return 0;
}
```

OUTPUT

Enter two numbers: 2.5 2.5
Enter task (1=add, 2=sub, 3=mul): 3
Result of operation is : 6.25

Explanation: In the above program, instead of calling functions directly, we have called them through function pointer pt. The complier is unable to use static or early binding in this case. In this program, the late binding concept is implemented. The compiler has to read the addresses held in the pointers toward different functions. Until run time, decisions are not taken as to which function needs to be executed; hence, it is late binding.

15.3 POINTER TO BASE AND DERIVED CLASS OBJECTS

In inheritance, the properties of existing classes are extended to the new classes. The new classes that can be created from the existing `base class` are called as `derived classes`. The inheritance provides the hierarchical organization of classes. It also provides the hierarchical relationship between two objects and indicates the shared properties between them. All derived classes inherit properties from the common base class. Pointers can be declared to the point base or derived class. Pointers to objects of the base class are type compatible with pointers to objects of the derived class. A base class pointer can point to objects of both the base and derived class. In other words, a pointer to the object of the base class can point to the object of the derived class; whereas a pointer to the object of the derived class cannot point to the object of the base class, as shown in Figure 15.2.

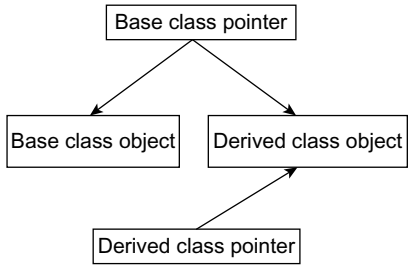

Fig. 15.2 Type compatibility of base and derived class pointers

15.4 Write a program to access members of base and derived class using pointer objects of both classes.

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

class W
{
       protected:
       int w;
       public:
       W (int k) { w=k; }
       void show()
       {
       cout<<"\n In base class W";
       cout<<"\n W="<<w; };
};
class X: public W
{
       protected:
       int x;
       public:
       X (int j, int k): W( j)
       {
       x=k;
       }
       void show()
       {
       cout<<"\n In class X";
       cout<<"\n w="<<w;
       cout<<"\n x="<<x;
       }
```

```
};
class Y: public X
{
      public:
      int y;
};
void main()
{
      clrscr();
      W *b;
      b = new W(20);   // pointer to class W
      b->show();
      delete b;
      b = new X(5,2);   // pointer to class X
      b->show();
      ((X*)b)->show();
      delete b;
      X x(3,4);
      X *d=&x;
      d->show();
}
```

OUTPUT

```
In base class W
W=20
In base class W
W=5
In class X
w=5
x=2
In class X
w=3
x=4
```

Explanation: In the above program, the class W is a base class. The X is derived from W, and class Y is derived from class X. Here, the type of inheritance is multilevel inheritance. The variable *b is a pointer object of the class W. The statement `b = new W (20);` creates a nameless object and returns its address to the pointer b. The `show()` function is invoked by the pointer object b. Here, the `show()` function of base class W is invoked. Using ***`delete`*** operator, the pointer b is deleted.

The statement `b = new X (5,2);` creates a nameless object of class X and assigns its address to the base class pointer b. Here, it should be noted that the object `b` is a pointer object of

the base class, and it is initialized with the address of the derived class object. Again, the pointer b invokes the function `show()` of the base class and not the function of class X (derived class.). To invoke the function of the derived class X, the following statement is used:

```
((X*)b)->show();
```

In the above statement, typecasting (upcasting) is used. The upcasting forces the object of class W to behave as if it were the object of class X. This time, the function `show()` of class X (derived class) is invoked. The process of obtaining the address of a derived class object, and treating it as the address of a base class object is known as `upcasting`.

The statement `X x(3,4);` creates object x of class X. The statement `X *d = &x;`

declares the pointer object d of the derived class X and assigns the address of x to it. The pointer object d invokes the derived class function `show()`. Figure 15.3 shows a pictorial representation of the above explanation.

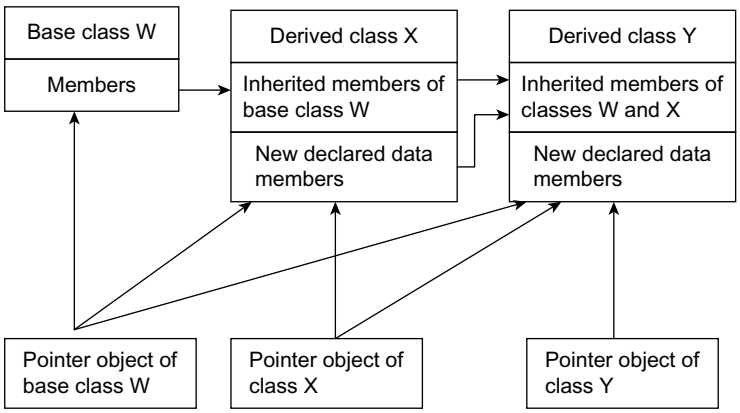

Fig. 15.3 Type compatibility of base and derived class pointers

Here, the pointer of class W points to its own class as well as its derived class. The pointer of derived class X can point to its own class but cannot point to its base class.

15.4 VIRTUAL FUNCTIONS

Virtual functions of the base class should be redefined in the derived classes. The programmer can define a virtual function in a base class, and can then use the same function name in any derived class, even if the number and type of arguments are matching. The matching function overrides the base class function of a similar name. Virtual functions can only be member functions. We can also declare the functions as given below:

`int` Base::get (`int`) and `int` Derived::get(`int`) even when they are not virtual.

The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of the object is available. With virtual functions, we cannot alter just the function type. It is illegal, therefore, to redefine a virtual function so that

it varies only in the return type. If two functions with a similar name have different arguments, C++ compiler considers them different, and the virtual function mechanism is dropped.

15.5 RULES FOR VIRTUAL FUNCTIONS

(1) The virtual function should not be static and must be a member of a class.
(2) The virtual function may be declared as a friend for another class. An object pointer can access the virtual functions.
(3) A constructor cannot be declared as virtual, but a destructor can be declared as virtual.
(4) The virtual function should be defined in the public section of the class. It is also possible to define the virtual function outside the class. In such a case, the declaration is done inside the class, and the definition is outside the class. The `virtual` keyword is used in the declaration and not in the function declarator.
(5) It is also possible to return a value from virtual functions similar to other functions.
(6) The prototype of the virtual function in the base class and derived class should be exactly the same. In case of a mismatch, the compiler neglects the virtual function mechanism and treats these functions as overloaded functions.
(7) Arithmetic operations cannot be used with base class pointers.
(8) If a base class contains a virtual function and if the same function is not redefined in the derived classes, in such a case, the base class function is invoked.
(9) The operator keyword used for operator overloading also supports the `virtual` mechanism.

15.5 Write a program to declare virtual function and execute the same function defined in the base and derived class.

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

class first
{
   int b;
   public:
   first() {b=10;}
   virtual void display() {cout<<"\n b=" <<b;}
};
class second : public first
{
   int d;
   public:
   second() {d=20;}
   void display() {cout<<"\n d="<<d;}
};
int main()
{
   clrscr();
```

```
    first f,*p;
    second s;
    p=&f;
    p->display();
    p=&s;
    p->display();
    return 0;
}
```

OUTPUT

b = 10
d = 20

Explanation: The above program is similar to the previous one. The only difference is that a `virtual` keyword precedes the `display()` function of the base class as per the statement `virtual void display() {cout<<"\n b = " <<b;}`. The `virtual` keyword does the run-time binding. In the first call, the `display()` function of the base class is executed and in the second call, that is, after assigning the address of the derived class to pointer p, `displays()` function of the derived class is executed.

15.6 Write a program to use pointer for both base and derived class and call the member function. Use virtual keyword.

VIRTUAL FUNCTIONS

```
#include<iostream.h>
#include<conio.h>
class super
{
   public:
   virtual void display() {cout<<"\n In function display() class
super";}
   virtual void show() {cout<<"\n In function show() class su-
per";}
}
;
class sub : public super
{
   public:
   void display() {cout<<"\nIn function display() class sub";}
   void show() {cout<<"\n In function show() class sub";}
};
int main()
{
```

```
    clrscr();
    super S;
    sub A;
    super *point;
    cout<<"\n Pointer point points to class super\n";
    point=&S;
    point->display();
    point->show();
    cout<<"\n\n Now Pointer point points to derived class sub\n";
    point=&A;
    point->display();
    point->show();
    return 0;
}
```

OUTPUT

Pointer point points to class super
In function display() class super
In function show() class super
Now Pointer point points to derived class sub
In function display() class sub
In function show() class sub

Explanation: In the above program, the `base class super` and the `derived class sub` have member functions a similar name. They are `display()` and `show()`. In function `main()`, the variable `S` is an object of `class super`, and the variable `A` is an object of derived `class sub`. The pointer variable `point` is a pointer to the base class. The address of object `S` is assigned to the pointer `S`. The pointer calls both the member functions. Similarly, the variable `A` is an object of the derived `class sub`. The address of `A` is assigned to the pointer `point` and again, the pointer calls the member functions.

The member functions of the base class are preceded by the keyword `virtual`. If the `virtual` keyword is removed, in both the function calls, the member functions of the base class are executed. The member functions for the derived class are not executed, though the pointer has the address of the derived class. If the `virtual` keyword is not removed, firstly, the member function of the base class is executed, and then member function of derived class.

15.6 ARRAY OF POINTERS

`Polymorphism` refers to late or dynamic binding; that is, the selection of an entity is decided at run time. In class hierarchy, methods with similar names can be defined, which perform different tasks, and then, the selection of the appropriate method is done using dynamic binding. Dynamic binding is associated with object pointers. Thus, addresses of different objects can be stored in an array to invoke functions dynamically. The following program explains this concept:

15.7 Write a program to create array of pointers. Invoke functions using array objects.

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

class A
{
   public:
   virtual void show()
   {
   cout<<"A\n";
   }
};
class B : public A
{
   public:
   void show() {cout<<"B\n";}
};
class C : public A
{
   public:
   void show()
   {cout<<"C\n";
   }
};
class D : public A
{
   public:
   void show()
   {
   cout<<"D\n";
   }
};
class E : public A
{
   public:
   void show()
   {
   cout<<"E";
   }
};
void main()
{
   clrscr();
   A a;
```

```
    B b;
    C c;
    D d;
    E e;
    A *pa[]={&a,&b,&c,&d,&e};
    for ( int j=0;j<5;j++)
    pa[j]->show();
}
```

OUTPUT

A
B
C
D
E

Explanation: In the above program, `class A` is a base class. The `classes B, C, D,` and `E` are classes derived from class A. All these classes have a similar function `show()`. In function `main()`, `a, b, c, d,` and `e` are objects of `classes A, B, C, D,` and `E,` respectively. The function `show()` of the base class is declared virtual. An array of pointer `*pa` is declared, and it is initialized with addresses of the base and derived class objects; that is, `a, b, c, d,` and `e`. Using `for` loop and array, each object invokes function `show()`. The output is as shown above. If the base class function `show()` is non-virtual, then the very time function `show()` of base class is executed. Figure 15.4 illustrates this concept more clearly.

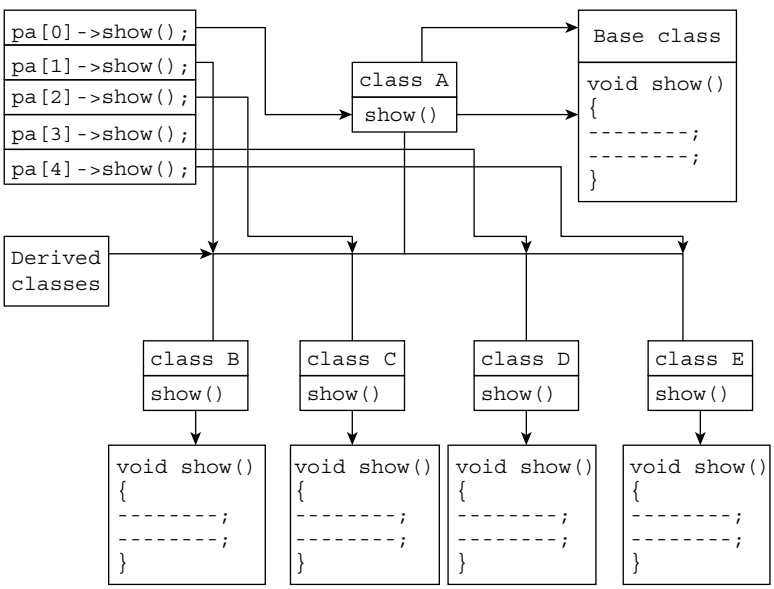

Fig. 15.4 Early and late binding of functions

15.7 PURE VIRTUAL FUNCTIONS

In practical applications, the member function of the base class is rarely used for doing any operation; such functions are called `do-nothing functions`, `dummy functions`, or `pure virtual functions`. The do-nothing functions or pure functions are always virtual functions. Usually, pure virtual functions are defined with a null body. This is so, because derived classes should be able to override them. Any normal function cannot be declared as a pure function. After the declaration of a pure function in a class, the class becomes an abstract class. It cannot be used to declare any object. Any attempt to declare an object will result in the error "`cannot create instance of abstract class`." The pure function can be declared as follows:

Declaration of pure virtual function

```
virtual void display() =0; // pure function
```

In the above declaration of the function, the `display()` is a pure virtual function. The assignment operator is not used to assign zero to this function. It is used just to instruct the compiler that the function is a pure virtual function and that it will not have a definition.

A pure virtual function declared in the base class cannot be used for any operation. The class containing the pure virtual function cannot be used to declare objects. Such classes are known as `abstract classes` or `pure abstract classes`. Anyone who attempts to declare an object from the abstract class would be reported an error message by the compiler. In addition, the compiler will display the name of the virtual function present in the base class. The classes derived from the pure abstract classes are required to re-declare the pure virtual function. All other derived classes without pure virtual functions are called `concrete classes`. The concrete classes can be used to create objects. A pure virtual function is similar to an unfilled container that the derived class is made to fill.

15.8 Write a program to declare pure virtual functions.

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

class first
{
   protected:
   int b;
   public:
   first() {b=10;}
   virtual void display() =0; // pure function
};
class second: public first
{
   int d;
   public:
   second() {d=20;}
```

```
    void display() {cout<<"\n b="<<b <<" d="<<d;}
};
int main()
{
    clrscr();
    first *p;
    // p->display // abnormal program termination
    second s;
    p=&s;
    p->display();
    return 0;
}
```

OUTPUT

b= 10 d = 20

Explanation: In the above program, the `display()` function of the base class is declared a pure function. The pointer object `*p` holds the address of the object of the derived class and invokes the function `display()` of the derived class. Here, the function `display()` of the base class does nothing. If we try to invoke the pure function using the statement `p->display()` as per the given remarks in the above program (`//p->display//abnormalprogram termination`), the program is terminated with the error "abnormal program termination."

15.8 ABSTRACT CLASSES

Abstract classes are similar to a skeleton on which new classes are designed to assemble a well-designed class hierarchy. The set of well-tested abstract classes can be used, and the programmer only extends them. Abstract classes containing virtual functions can be used to help in program debugging. When various programmers work on the same project, it is necessary to create a common abstract base class for them. The programmers are restricted to create a new base class.

The development of such software can be demonstrated by creating a header file. The file `abstract.h` is an example of a file containing an abstract base class that is used for debugging purpose. Contents of the file `abstract.h` are as follows:

Contents of abstract.h header file

```
#include<iostream.h>
struct debug
{
    virtual void show()
    {
    cout<<"\n No function show() defined for this class";
    }
};
```

15.9 Write program to use abstract class for program debugging.

```
#include"abstract.h"
#include<constream.h>

class A : public debug
{
   int a;
   public:

   A(int j=0) {a=j;
}
void show() {cout<<"\nIn class A a="<<a;}
};
class B : public debug
{
   int b;
   public:
   B (int k ) {b=k;}
};
void main()
{
   clrscr();
   A a(1);
   B b(5);
   a.show();
   b.show();
}
```

OUTPUT

In class A a=1
No function show() defined for this class

Explanation: Observe the contents of file abstract.h. The struct debug contains the virtual function `show()`. This function is declared in the file `abstract.h` and the same is inserted using `#include` directive in the above program. The classes A and B are derived from class debug, which is defined in the header file abstract.h. In function, a and b are the objects of classes A and B, respectively. The statement `a.show();` invokes the function `show()`, and the value of a is displayed. The object b also invokes the function `show()`. However, the class B does not have the function `show()`. Hence, the function `show()` of an abstract base class is executed, which displays a warning message.

Traditional languages do not provide such facilities. An abstract class develops into a dominant and powerful interface when the software system undergoes various changes. It is essential

to confirm that the debugging interface is accurately constructed. If changes are made in the actual project, it is compulsory to add appropriate methods to the abstract base class. In case the programmer needs to define a function `warn()` to the class debug, then the header file can be updated. The contents of the file would be as follows:

Contents of abstract.h header file

```
#include<iostream.h>
struct debug
{
   virtual void show()
   {
   cout<<"\n No function show() defined for this
class";
   }
};
virtual void warn()
{
   cout<<"\n No function warn() defined for this
class";
}
};
```

While defining such an abstract class, the following points should be kept in mind:

(1) Do not declare an object of abstract class type.
(2) An abstract class can be used as a base class.
(3) The derived class should not have pure virtual functions. Objects of the derived class can be declared.

15.9 WORKING OF VIRTUAL FUNCTIONS

Before learning about the mechanism of virtual functions, let us revise a few points related to virtual functions:

(1) `Binding` means a link between a function call and the real function that is executed when the function is called.
(2) When a compiler knows which function to call before execution, it is known as `early binding`.
(3) `Dynamic binding` means the actual function invoked at run time is dependent on the address stored in the pointer. In this binding, a link between function call and actual function is made during program execution.
(4) The keyword `virtual` prevents the compiler from performing early binding. Binding is postponed until program execution.

The following programs illustrate the step-by-step working of virtual functions:

15.10 Write a program to define virtual and non-virtual functions and determine the size of the objects.

```
#include<iostream.h>
#include<conio.h>
class A
{
   private:
   int j;
   public:
   virtual void show() {cout<<endl<<"In A class";}
};
class B
{
   private:
   int j;
   public:
   void show() {cout<<endl<<"in B class";}
};
class C
{
   public:
   void show() {cout<<endl<<"In C class";}
};
void main()
{
   clrscr();
   A x;
   B y;
   C z;
   cout<<endl<<"Size of x="<<sizeof (x);
   cout<<endl<<"Size of y="<<sizeof (y);
   cout<<endl<<"Size of z="<<sizeof (z);
}
```

OUTPUT

Size of x = 4
Size of y = 2
Size of z = 1

Explanation: In the above program, the `class A` has only one data element of integer type, but the size of the object displayed is `4`. The size of the object of `class B` that contains a one-integer data element is `two`**,** and finally, the size of the object of `class C` is displayed as `one`, even if the `class C` has no data element.

The function `show()` of `class A` is prefixed by the `virtual` keyword. Without the `virtual` keyword, the size of objects would be 2,2 and 1. The size of object x with a virtual function in `class A` is the addition of data member `int` (2 bytes) and void pointers (2 bytes). When a function is declared as virtual, the compiler inserts a void pointer. In class C, even if it has no object, the size of the object displayed is 1, and this is due to the compiler, who assumes the size of object z not to be zero, as every object should have an individual address. Hence, minimum size one is considered. The minimum nonzero positive integer is one.

To perform late binding, the compiler establishes VTABLE (virtual table) for every class and its derived classes having a virtual function. The VTABLE contains addresses of the virtual functions. The compiler puts the address of the virtual functions in the VTABLE. If no function is redefined in the derived class that is defined as `virtual` in the base class, the compiler takes the address of the base class function.

When an object of the base or derived class is created, a void pointer is inserted in the VTABLE, called *VPTR* (vpointer). The VPTR points to the VTABLE. When a virtual function is invoked, using the base class pointer, the compiler speedily puts a code to obtain the VPTR and searches for the address of the function in the VTABLE. In this way, an appropriate function is invoked, and dynamic binding takes place.

The VPTR should be initialized with the beginning address of the apposite VTABLE. When the VPTR is initialized with the apposite VTABLE, the type of the object can be determined by itself. However, it is useless if is applied at the point when a virtual function is invoked.

Assume a base class pointer object points to the object of a derived class. If a function is invoked using the base class pointer, the compiler uses a different code to accomplish the function call. The compiler begins from the base class pointer. The base class pointer holds the address of the derived class object. With the aid of this address, the VPTR of the derived class is obtained. Via VPTR, the VTABLE of the derived class is obtained. In the VTABLE, the address of the function being invoked is acquired and the function is called. All the above processes are handled by the compiler, and the user need not worry about them. The following program makes the concept clearer:

15.11 Write a program to define virtual member functions in derived classes.

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

class shop
{
   public:
   virtual void area() {cout<<endl<<"In area of shop";}
   virtual void rent() {cout<<endl<<"In rent of shop";}
```

```
   void period() {cout<<endl<<"In period of shop";}
};
class shopA : public shop
{
   public:
   void area() {cout<<endl<<"In area of shopA";}
   void rent() {cout<<endl<<"In rent of shopA";}
};
class shopB : public shop
{
   public:
   void area() {cout<<endl<<"In area of shopB";}
   void rent() {cout<<endl<<"In rent of shopB";}
   void period() {cout<<endl<<"In period of shopB";}
};
class shopC: public shop
{
   void area() {cout<<endl<<"In area of shopC";}
};
void main()
{
   clrscr();
   shop *p;
   shop s;
   p=&s;
   p->area();
   p->rent();
   p->period();
   shop *sa,*sb,*sc;
   shopA a;
   shopB b;
   shopC c;
   sa=&a;
   sb=&b;
   sc=&c;
   sa->area();
   sa->rent();
   sb->area();
   sb->rent();
   sc->area();
   sc->rent();
   sa->period();
```

```
    e.area();
    sb->period();
    shop d;
    d.area();
    shopA e;
    shopC f;
    f.rent();
}
```

OUTPUT

```
In area of shop
In rent of shop
In period of shop
In area of shopA
In rent of shopA
In area of shopB
In rent of shopB
In area of shopC
In rent of shop
In period of shop
In period of shop
In area of shop
In area of shopA
In rent of shop
```

Explanation: In the above program, four classes are declared, as shown in Figure 15.5.

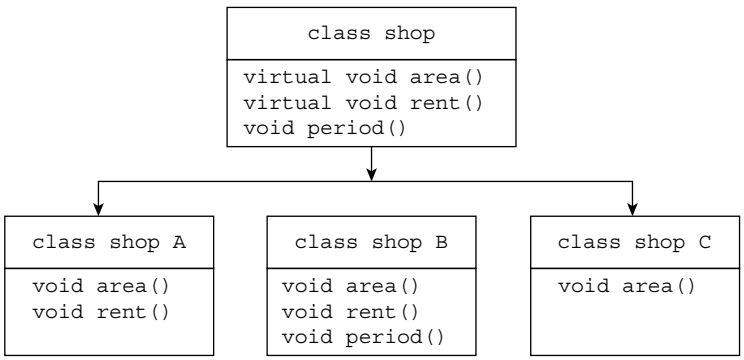

Fig. 15.5 Base and derived classes

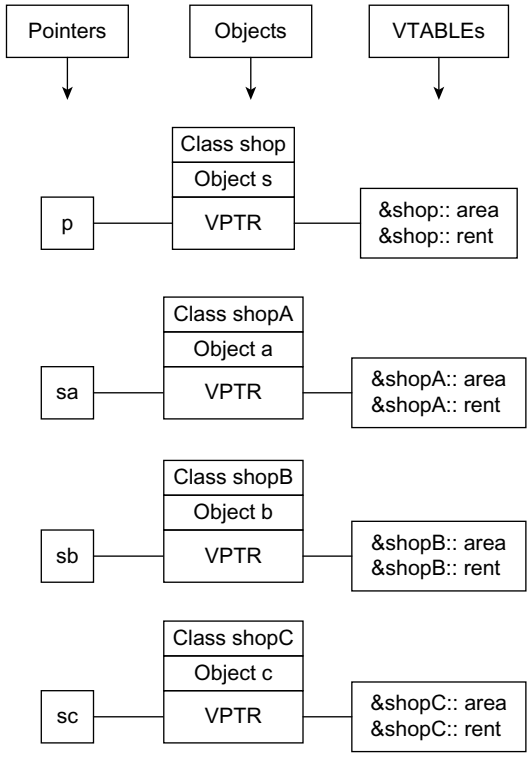

Fig. 15.6 VPTR AND VTABLES

As discussed earlier, a VTABLE is formed for each class having a virtual function and for the derived class of the same class. The VTABLE is formed for the following classes: shop, shopA, shopB, and shopC. All of the above four VTABLES hold the address of virtual functions. In addition, the compiler would place a VPTR that points to the particular VTABLE, as shown in Figure 15.6.

The `class shopC` is without functions `rent()`. Therefore, the VTABLE holds the address of the base class `rent()` function. Consider the following statements:

```
shop *p; // Base class object pointer
shop s; // object of base class
p = &s; // Address of s is assigned to p
```

The pointer `p` contains the address of the object `s`. Now, consider the following statements:

```
p->area(); // Invokes function area() of base class
p->rent(); // Invokes function rent() of base class
p->period(); // Invokes function period() of base class
```

From the above functions, first two function `area()` and `rent()` are virtual, and `period()` is a non-virtual function. Though the address of the base class object or derived class object is stored in the base class pointer, the function of the base class is invoked, because the `period()` is a non-virtual function. The member function `area()` is declared as virtual in the base class. All the three derived classes shopA, shopB, and shopC contain the function `area()`. The execu-

tion of these functions depends on the address stored in the base class pointer. For example, p contains the address of object a; the functions of class shopA are invoked. Similarly, for storing addresses of objects b and c, functions of classB and classC can be invoked.

p = &s; // Address of s is assigned to p

In the above statement, `p` contains the address of the object `s` (base class). Therefore, when the function `area()` is invoked by the pointer `p`, VPTR is created from the object `s`. With the help of VPTR, VTABLE of the class shop is obtained, and the address of the function `area()` for class shop is accessed. With the aid of an address, `shop::area()` is finally invoked. Consider the following statements:

***shop *sa;** // object pointer declaration*
***shopA a;** // object of derived class*
***sa = &a;** // assigns address of derived class object to base class pointer*
***sa->area();** // invokes function area()*
***sa->rent();** // invokes function rent()*

In the statement `sa = &a;` the address of the derived class object is stored in the base class pointer. When the function `area()` is invoked, the VPTR of object a is used to obtain the VTABLE of class shopA. From this VTABLE, the address of `shopA::area()` and `shopA::rent` are obtained and finally invoked. Likewise, for objects b and c, binding is achieved.

Consider the following statements:

sa->period();
sb->period();

Function `period()` is a non-virtual function. The VTABLE is not used to call the function `shop::period()`. In addition, the function `period()` is not redefined in the derived classes. Now, concentrate on the following statements:

shop d; // Base class object
d.area();
shopA e; // Derived class object
e.area();
shopC f; // Derived class object
f..rent();

In the above statements, dynamic binding is not performed, and, hence, VPTR and VTABLE are not created. The functions are called as usual.

C++ places addresses of the virtual member function in the virtual table. When these member functions are called, the accurate address is obtained from the virtual table. This entire procedure takes time. Therefore, the virtual function makes program execution a bit slow. Conversely, it provides better flexibility.

15.10 VIRTUAL FUNCTIONS IN DERIVED CLASSES

We know that when functions are declared virtual in the base class, it is mandatory to redefine virtual functions in the derived class. The compiler creates VTABLES for the derived class and stores addresses of functions in it. In case the virtual function is not redefined in the derived class, the VTABLE of the derived class contains the address of the base class virtual function. Thus, the

VTABLE contains addresses of all functions. Thus, it is not possible for a function to exist but for its address to not be present in the VTABLE. Consider the following program, which shows the possibility of a function existing but an address not being located in the VTABLE:

15.12 Write a program to redefine a virtual base class function in the derived class. Also, add a new member in the derived class. Observe the VTABLE.

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

class A
{
   public:
   virtual void joy()
   {
   cout<<endl<<"In joy of class A";
   }
};
class B: public A
{
   public:
   void joy() {cout<<endl<<"In joy of class B";}
   void virtual joy2()
   {
   cout<<endl<<"In joy2 of class B";
   }
};
void main()
{
   clrscr();
   A *a1,*a2;
   A a3;
   B b;
   a1=&a3;
   a2=&b;
   a1->joy();
   a2->joy();
   // a2->joy2(); // joy2 is not member of class A
}
```

OUTPUT

In joy of class A
In joy of class B

Explanation: In the above program, the base class A contains one virtual function `joy()`. In the derived class B, the function `joy()` is redefined and defines a new virtual function `joy2()`. Figure 15.7 shows VTABLES created for the base and derived class.

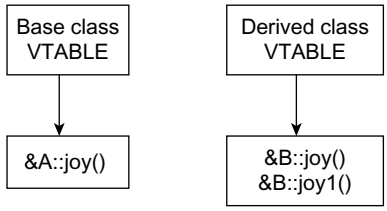

Fig. 15.7 VTABLES for base and derived class

```
a2->joy2();
```

The above statement will generate an error message. In the above statement, the pointer a2 is treated as only a pointer to the base class object. The function `joy2()` is not a member of base class A. Hence, it is not allowed to invoke the function `joy2()` using the pointer object a2. However, an address of the derived class is assigned to the base class pointer, and the compiler has no way to determine that we are working with a derived class object. The compiler avoids calling virtual functions present only in derived classes. In a hierarchical class organization of various levels, if it is essential to invoke a function at any level by using a base class pointer, then the function should be declared virtual in the base class.

15.11 OBJECT SLICING

Virtual functions permit us to manipulate both base and derived objects using similar member functions with no modifications. Virtual functions can be invoked using a pointer or reference. If we do so, object slicing takes place. The following program takes you to the real thing:

15.13 Write a program to declare reference to object and invoke functions.

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

class B
{
   int j;
   public:
   B (int jj) {j=jj;}
   virtual void joy()
   {
   cout<<"\n In class B";
   cout<<endl<<" j="<<j;
   }
};
class D : public B
{
   int k;
   public:
   D (int jj, int kk) : B (jj)
```

```
    {k=kk;}
    void joy()
    {
    B::joy();
    cout<<"\n In class D";
    cout<<endl<<" k="<<k;
    }
};
void main()
{
    clrscr();
    B b(3);
    D d (4,5);
    B &r=d;
    cout<<"\n Using Object";
    d.joy();
    cout<<"\n Using Reference";
    r.joy();
}
```

OUTPUT

```
Using Object
In class B
j= 4
In class D
k= 5
Using Reference
In class B
j= 4
In class D
k= 5
```

Explanation: In the above program, a reference object r is created to an object d using the statement `B &r = d;`. The member function `joy()` is invoked using r and d objects. The output is similar.

15.14 Write a program to demonstrate object slicing.

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

class A
{
   public:
   int a;
```

```
   A() {a=10;}
};
class B: public A
{
   public:
   int b;
   B() {a=40; b=30;}
};
void main()
{
   clrscr();
   A x;
   B y;
   cout<<" a="<<x.a <<"";
   x=y;
   cout<<" Now a="<<x.a;
}
```

OUTPUT

a=10 now a=40

Explanation: In the above program, class A has only one data member a and derived class B has one member b. In function `main()`, objects x and y of classes A and B are declared. The object x has only one member, that is, a and the object y has two members a and b. The statement x = y, that is, the derived class object, is assigned to the base class object. In such an assignment, only base, class part of the derived object is assigned to the base class object. Thus, if an object of a derived class is assigned to a base class object, the compiler allows it. However, it copies only the base class members of the object, and this process is known as `object slicing`.

15.12 CONSTRUCTORS AND VIRTUAL FUNCTIONS

It is possible to invoke a virtual function using a constructor. A constructor makes the virtual mechanism illegal. When a virtual function is invoked through a constructor, the base class virtual function will not be called; instead, the member function of a similar class is invoked.

15.15 Write a program to call virtual function through constructor.

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

class B
{
   int k;
   public:
   B (int l) {k=l;}
```

```
   virtual void show() {cout<<endl<<" k="<<k;}
};
class D: public B
{
   int h;
   public:
   D (int m, int n) : B (m)
   {
   h=n;
   B *b;
   b=this;
   b->show();
   }
   void show()
   {
   cout<<endl<<" h="<<h;
   }
};
void main()
{
   clrscr();
   B b(4);
   D d(5,2);
}
```

OUTPUT

h=2

Explanation: In the above program, the base class B contains a virtual function. In the derived class D, a similar function is redefined. Both the base and derived classes contain a constructor. In the derived class constructor, the base class pointer *b is declared. We know that this pointer contains the address of the object calling the member function. Here, the `this` pointer holds the address of the object d. The pointer object b invokes the function `show()`. The derived class `show()` function is invoked.

Here, the object d is not fully constructed; then, how does it invoke the member function of a similar class? This is possible, because a virtual function call reaches ahead into inheritance.

15.13 VIRTUAL DESTRUCTORS

We have learned how to declare virtual functions. Likewise, destructors can be declared as virtual. The constructor cannot be virtual, as it requires information about the accurate type of the object in order to construct it properly. The virtual destructors are implemented in a similar manner to virtual functions. In constructors and destructors, pecking order (hierarchy) of base and

derived classes is constructed. Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

For example, a derived class object is constructed using a new operator. The base class pointer object holds the address of the derived object. When the base class pointer is destructed using the delete operator, the destructor of the base and derived class is executed. The following program explains this:

15.16 Write a program to define virtual destructors.

```
#include<iostream.h>
#include<conio.h>
class B
{
   public:
   B() {cout<<endl<<"In constructor of class B";}
   virtual ~B() {cout<<endl<<"In destructor of class B";}
};
class D: public B
{
   public:
   D() {cout<<endl<<"In constructor of class D";}
   ~ D() {cout<<endl<<"In destructor of class D";}
};
void main()
{
   clrscr();
   B *p;
   p= new D;
   delete p;
}
```

OUTPUT

```
In constructor of class B
In constructor of class D
In destructor of class D
In destructor of class B
```

Explanation: In the above program, the destructor of the base class B is declared as virtual. A dynamic object is created, and the address of the nameless object that is created is assigned to pointer p. The new operator allocates the memory required for data members. When the object goes out of scope, it should be deleted, and the same should be performed by the statement delete *p;*. When the derived class object is pointed by the base class pointer object, in order to invoke the base class destructor, virtual destructors are useful.

15.14 DESTRUCTORS AND VIRTUAL FUNCTIONS

When a virtual function is invoked through a non-virtual member function, late binding is performed. When a virtual function is called through the destructor, the redefined function of a similar class is invoked. Consider the following program:

15.17 Write a program to call virtual function using destructors.

```
#include<iostream.h>
#include<conio.h>
class B
{
   public:
   ~ B() {cout<<endl<<" in virtual destructor";}
   virtual void joy() {cout<<endl<<"In joy of class B";}
};
class D : public B
{
   public :
   ~ D()
   {
   B *p;
   p=this;
   p->joy();
   }
   void joy() {cout<<endl<<" in joy() of class D";}
};
void main()
{
   clrscr();
   D X;
}
```

OUTPUT

```
in joy() of class D
in virtual destructor
```

Explanation: In the above program, the destructor of the derived class function joy() is invoked. The member function joy() of the derived class is invoked followed by the virtual destructor.

SUMMARY

(1) The word `poly` means `many`, and `morphism` means `several` forms. Both the words are derived from Greek language. Thus, by combining these two words, a new whole word called `polymorphism` is created, which means `various forms`.

(2) The information pertaining to various overloaded member functions is to be given to the compiler while compiling. This is called `early binding` or `static binding`. The deciding function call at run time is called `run time` or `late` or `dynamic binding`. Dynamic binding permits to suspend the decision of choosing a suitable member function until run time.

(3) Pointers to the object of a base class are type compatible with pointers to the object of a derived class. The reverse is not possible.

(4) Virtual functions of the base class should be redefined in the derived classes. The programmer can define a virtual function in a base class, and can then use a similar function name in any derived class.

(5) Addresses of different objects can be stored in an array to invoke the function dynamically.

(6) In practical applications, the member function of the base class is rarely used for doing any operation; such functions are called `do-nothing functions`, `dummy functions`, or `pure virtual functions`.

(7) All other derived classes without pure virtual functions are called `concrete classes`.

(8) Abstract classes are similar to a skeleton on which new classes are designed to assemble a well-designed class hierarchy. They are not used for object declaration.

(9) Virtual functions can be invoked using a pointer or a reference.

(10) If an object of a derived class is assigned to a base class object, the compiler allows it. However, it copies only the base class members of the object, and this process is known as `object slicing`.

(11) It is possible to invoke a virtual function using a constructor. The constructor makes the virtual mechanism illegal.

(12) We have learned how to declare virtual functions. Likewise, destructors can be declared as virtual. The constructor cannot be virtual. The virtual destructors are implemented in a similar manner to virtual functions. Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

EXERCISES

(A) Answer the following questions

(1) What is polymorphism?
(2) Explain compile-time and run-time binding.
(3) Explain the use of a virtual keyword.
(4) What are pure functions? How are they declared?
(5) Is it possible to declare an object of the class that contains a pure function?
(6) What is the difference between a virtual function and virtual classes?
(7) What are virtual destructors?
(8) How does a C++ compiler accomplish dynamic binding?
(9) Where do we use virtual functions? Give their applications.

(10) What is early binding and late binding?
(11) Explain object slicing.
(12) Explain virtual destructors.
(13) What are abstract classes? How can they be used for a debugging program?
(14) What are VPTR and VTABLE? Explain in detail.
(15) Describe the rules for declaring virtual functions.
(16) What is the difference between a base class pointer and a derived class pointer?

(B) Answer the following by selecting the appropriate option

(1) Consider the statement `virtual void display()=0`. The `display()` function is
  (a) pure virtual function
  (b) pure member function
  (c) normal function
  (d) all of the above
(2) The do-nothing function is nothing but
  (a) pure virtual function
  (b) pure member function
  (c) both (a) and (b)
  (d) none of the above
(3) Static binding is done
  (a) at the time of compilation of the program
  (b) at run time
  (c) both (a) and (b)
  (d) none of the above
(4) Dynamic binding is done using the keyword
  (a) virtual
  (b) inline
  (c) static
  (d) void
(5) The `virtual` keyword solves the
  (a) ambiguity in the base and derived class
  (b) ambiguity in derived classes
  (c) ambiguity in base classes
  (d) none of the above
(6) B is a base class object, and D is a derived class object. The statement B = D
  (a) copies all elements of object D to object B
  (b) copies only the base portion of object D to B
  (c) copies only the derived portion of object D to B
  (d) none of the above
(7) When a base class is not used for objects, the declaration is called
  (a) abstract class
  (b) container class
  (c) concrete class
  (d) derived class
(8) The derived class without a pure virtual function is called
  (a) concrete class
  (b) abstract class
  (c) container class
  (d) derived class
(9) A pointer to base class object can hold addresses of
  (a) only derived class objects
  (b) only base class objects
  (c) addresses of a base class object and its derived class object
  (d) none of the above

(C) Attempt the following programs

(1) Write a program to declare a function `show()` in base and derived classes. Display a message through the function to know the name of the class whose member function is executed. Use the late binding concept using the `virtual` keyword.
(2) Write a program to define classes A, B, and C. The class C is derived from classes A and B. Define `count()` member function in all the classes as virtual. Count the number of objects created.
(3) Write a program to declare a matrix class, which has a data-member integer array as 3×3. Derive class matrix A from class matrix B and matrix B from matrix A. All these classes should have a function `show()` to display the contents. Read and display the elements of all three matrices.

(4) Write a program to declare the reference object. Invoke the member function of the class using the reference object.
(5) Write a program to demonstrate object slicing.
(6) Write a program to demonstrate the use of an abstract class.
(7) Write a program to redefine a virtual base class function in the derived class. In addition, add a new member in the derived class. Observe the VTABLE.
(8) Write a program to define virtual, non-virtual functions and determine the size of the objects.
(9) Write a program to invoke the member function of the base and derived class using the pointer of the base class.
(10) Write a program to access the members of the base and derived class using the pointer objects of both classes.

(D) Find the bugs in the following programs

(1)

```
class B {virtual void
display()=0;};
void main() {B d;}
```

(2)

```
class B
{
public:
virtual void display()
{
cout<<"\n no function display
defined in this class.";
}
};
struct D : B {};
void main()
{
D d;
d.display();
}
```

(3)

```
class B
{
   int a,b,c;
   public:
   B() {a=10, b=20,c=40;}
};
class D : public B {};
void main()
{
   B b;
   D d;
   d=b;
}
```

(4)

```
class B {};
class D : public B
{
   int i,j,k;
   public:
   D() {i=5; j=10; k=15;}};
void main()
{
   B b;
   D d;
   b=d;
   cout<<b.i;
}
```

(5)

```
class B
{public:
   B()  {cout<<endl<<"In  con-
structor of class B";}
   virtual ~B() =0;
};
class D : public B
{
   public:
   D() {cout<<endl <<"In con-
structor of class D";}
   ~D()  {cout<<endl<<"In  de-
structor of class D";}
};
void main()
{
   B *p;
   p=new D;
   delete p;
}
```