Virtual function
&
Run time polymorphism

Time to think!

What happens to overrridden functions accessed by base class pointer?

```
class Base
   {
   public:
      void show()
         { cout << "Base\n"; }
   };
//////////////////////////////
class Derv1 : public Base
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
```

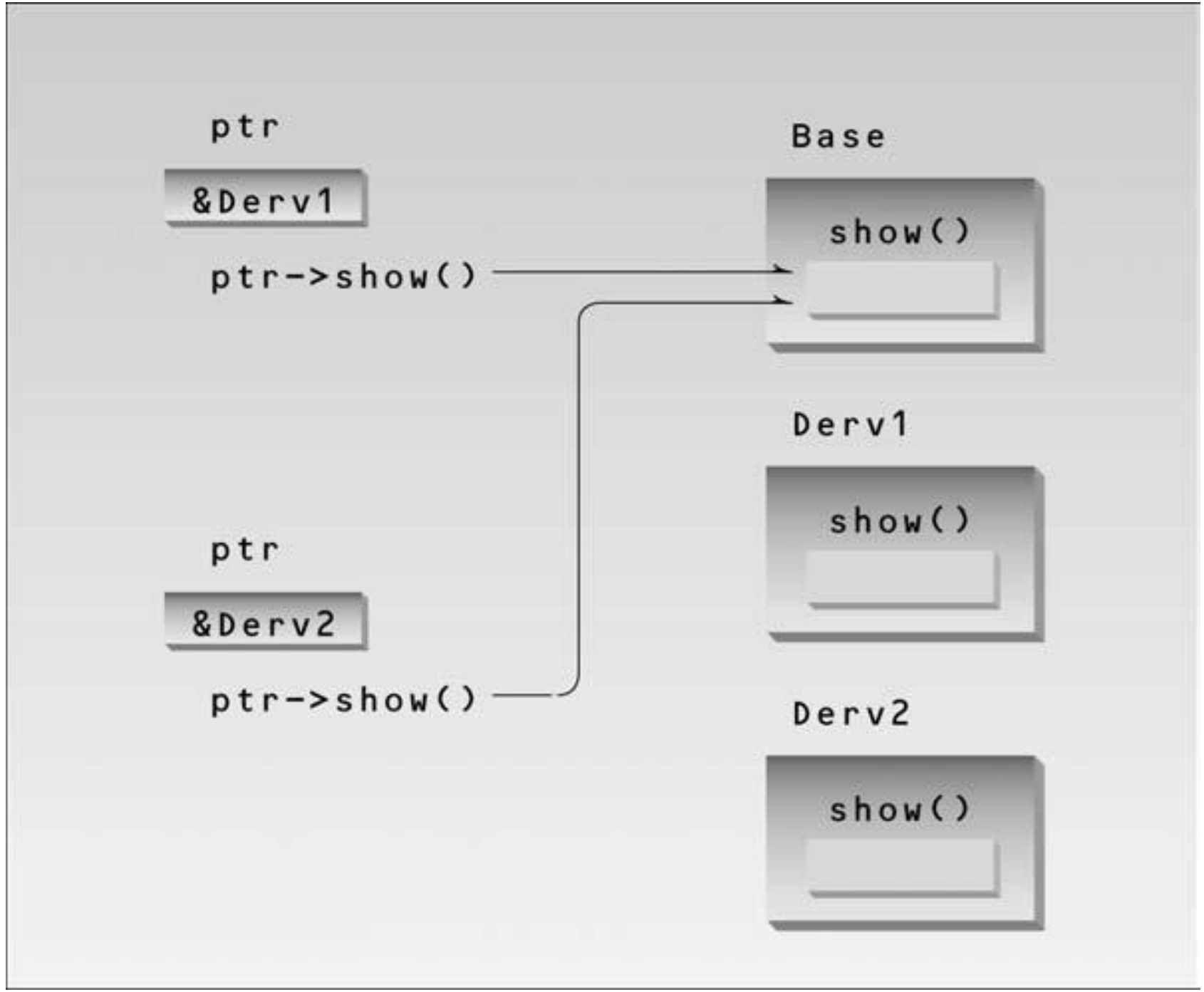
```
class Derv2 : public Base
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
```

```
int main()
   {
   Derv1 dv1;              //object of derived class 1
   Derv2 dv2;              //object of derived class 2
   Base* ptr;              //pointer to base class

   ptr = &dv1;             //put address of dv1 in pointer
   ptr->show();            //execute show()
   ptr = &dv2;             //put address of dv2 in pointer
   ptr->show();            //execute show()
   return 0;
   }
```

OUTPUT

```
Base
Base
```

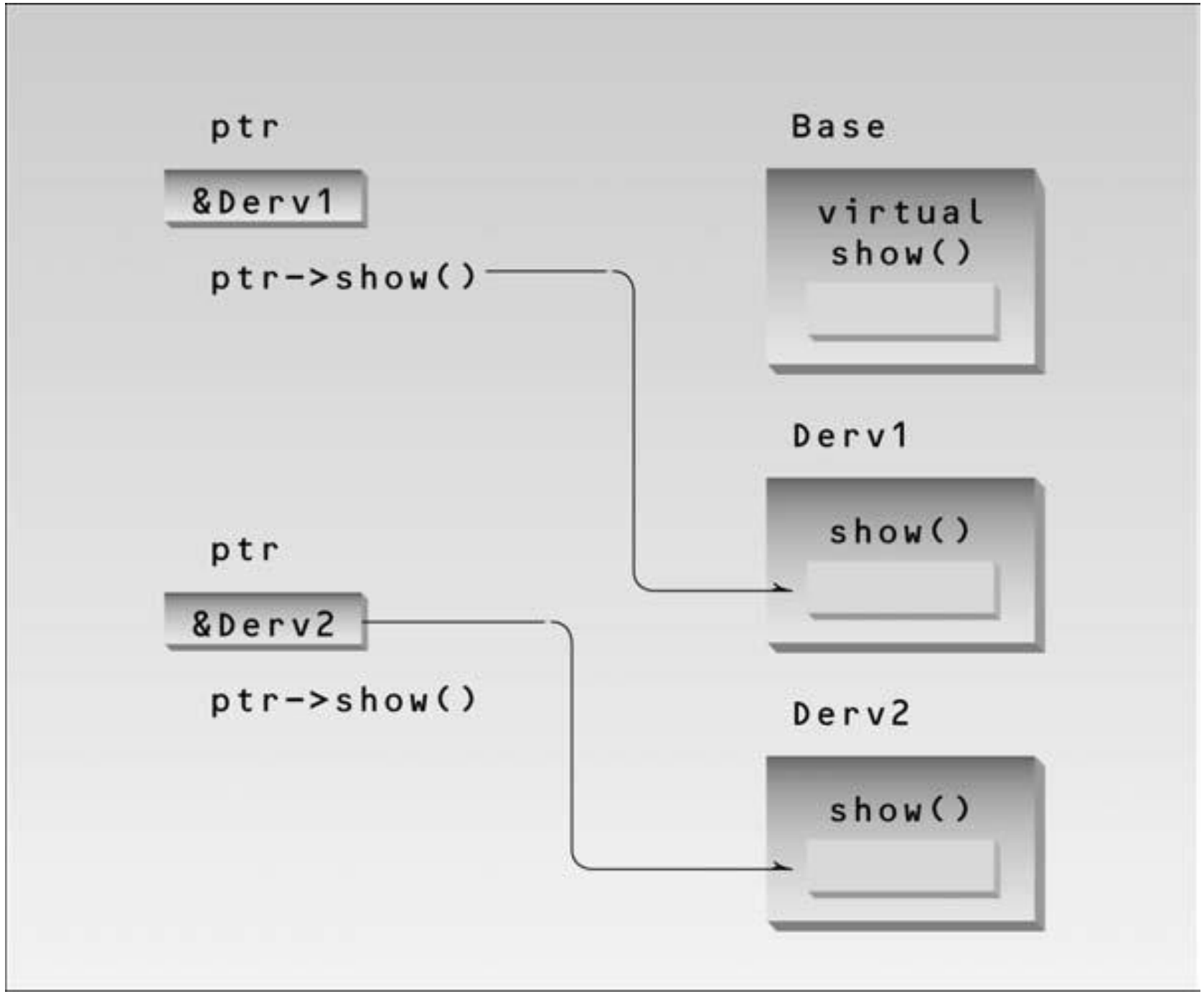

ptr
&Derv1
ptr->show()
Base
show()
Derv1
show()
ptr
&Derv2
ptr->show()
Derv2
show()

Virtual function and Run-Time polymorphism:

```
3  class Base
4  {
5  public:
6      virtual void show() //virtual function
7      {
8          cout<<"Base\n";
9      }
10 };
11
12 class Derv1 : public Base
13 {
14 public:
15     void show()
16     { cout <<"Derv1\n"; }
17 };
19 class Derv2 : public Base
20 {
21 public:
22     void show()
23     { cout <<"Derv2\n"; }
24 };
```

```
26 int main()
27 {
28     Base* bptr; //pointer to base class
29     Derv1 dv1;
30     Derv2 dv2;
31     bptr = &dv1;  //put address of dv1 in bptr
32     bptr->show(); //execute show()
33     bptr = &dv2;  //put address of dv2 in bptr
34     bptr->show(); //execute show()
35 }
```

```
Derv1
Derv2
```


ptr
&Derv1
ptr->show()
Base
virtual
show()
Derv1
show()
ptr
&Derv2
ptr->show()
Derv2
show()

Question-1:

```
3   class Base
4   {
5   public:
6       virtual void show1()
7       { cout<<"show1() of Base\n"; }
8       void show2()
9       { cout<<"show2() of Base\n"; }
10  };
11
12  class Derv1 : public Base
13  {
14  public:
15      void show1()
16      { cout <<"show1() of Derv1\n"; }
17
18      void show2()
19      { cout <<"show2() of Derv1\n"; }
20  };
```

```
22  int main()
23  {
24      Base *bptr = new Derv1;
25      bptr->show1();
26      bptr->show2();
27  }
```

```
show1() of Derv1
show2() of Base
```

Time to think!

What happens to non-overrridden functions accessed by base class pointer?

```
3   class Base
4   {
5   public:
6       virtual void show1()
7       { cout<<"show1() of Base\n"; }
8   };
9
10  class Derv : public Base
11  {
12  public:
13      void show1()
14      { cout <<"show1() of Derv1\n"; }
15
16      void show2()
17      { cout <<"show2() of Derv1\n"; }
18  };
```

```
20  int main()
21  {
22      Base *bptr;
23      Derv D;
24      bptr = &D;
25      bptr->show1();
26      bptr->show2(); //Error
27  }
```

Only the inherited features are accessible through base class pointer

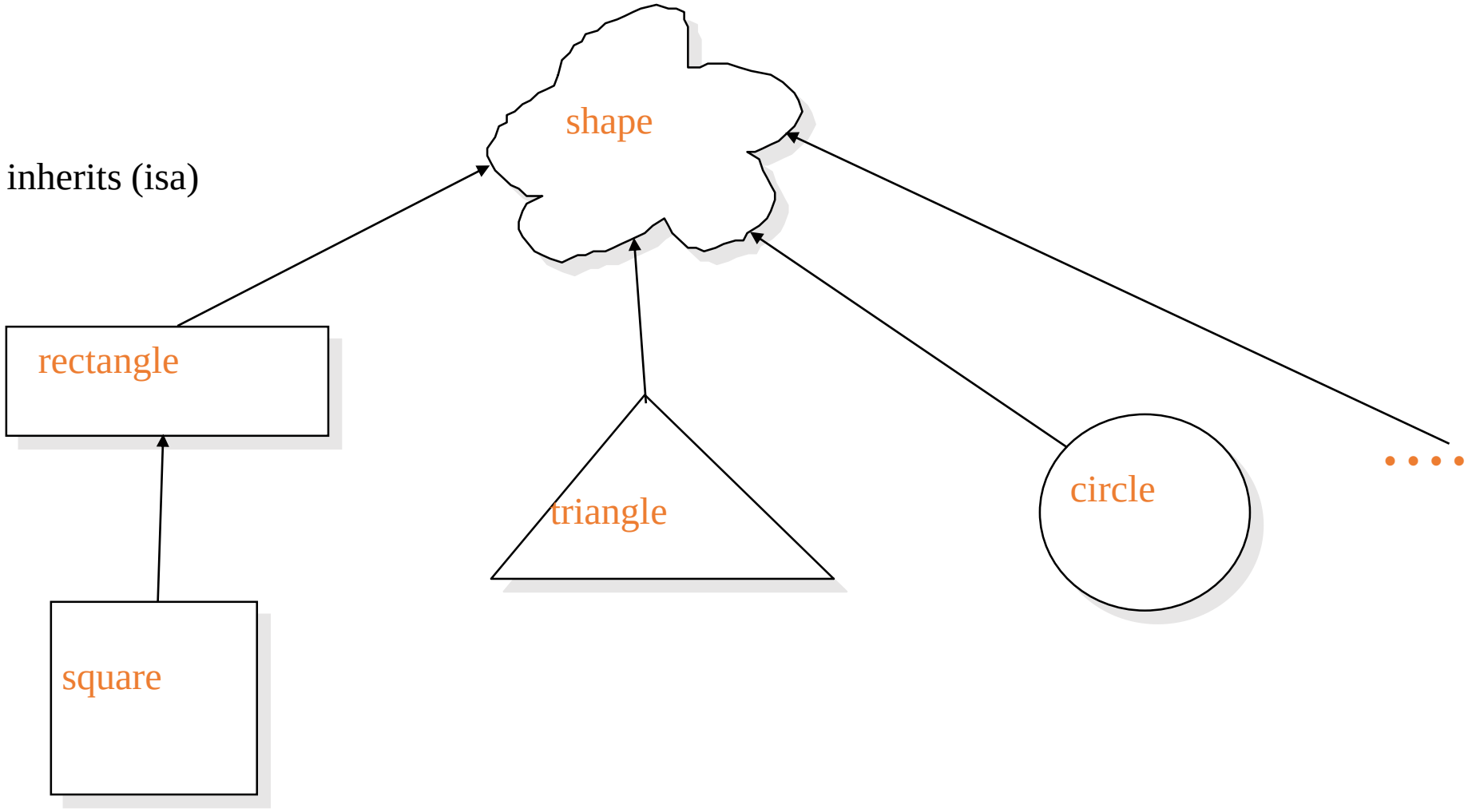
Pure virtual function & abstract class

```
3   class Base
4   {
5   public:
6         virtual void show() = 0;
7   };
8
9   class Der1 : public Base
10  {
11  public:
12      void show()
13      {  cout<<"\n show() of Der1..";  }
14  };
15
16  class Der2 : public Base
17  {
18  public:
19      void show()
20      {  cout<<"\n show() of Der2..";  }
21  };
```

```
23  int main()
24  {
25      Base * Base_ptr;
26      Der1 D;
27      // Base B; // Error
28      Base_ptr = &D;
29      Base_ptr->show();
30      Base_ptr = new Der2();
31      Base_ptr->show();
32  }
```

```
show() of Der1..
show() of Der2..
```

Representing Shapes



```
3   const float PI = 22.0/7.0;
4   class Shape
5   {
6   public:
7         virtual double area() = 0;
8   };
9
10  class Rectangle: public Shape
11  {
12      double height, width;
13  public:
14      Rectangle(float h, float w)
15      { height = h; width = w; }
16
17      double area()
18      {  return height*width; }
19  };
```

```
20  class Circle: public Shape
21  {
22      double radius;
23  public:
24      Circle(int r)
25      { radius = r; }
26
27      double area()
28      { return PI*radius*radius; }
29  };
```

```
31  int main()
32  {
33      Circle c1(10);
34      Rectangle r1(10,20);
35
36      int No_of_objs = 4;
37      Shape* p[] = { &c1, new Rectangle(5,6), &r1, new Circle(100)};
38      for (int i = 0; i < No_of_objs; ++i)
39          cout<<"\nArea = "<<p[i]->area();
40      return 0;
41  }
```

Each element of the array is an address of derived class object

```
Area = 314.286
Area = 30
Area = 200
Area = 31428.6
```

Question:

```
3   class Base
4   {
5   public:
6         virtual void show() = 0;
7   };
8
9   class Der1 : public Base
10  {
11  public:
12      void show()
13      {  cout<<"\n show() of Der1..";  }
14  };
15
16  class Der2 : public Base
17  { };
```

```
19  int main()
20  {
21      Base * Base_ptr;
22      Der1 D1;
23      Der2 D2;        //ERROR
24      Base_ptr = &D1;
25      Base_ptr->show();
26      Base_ptr = new Der2();  //ERROR
27      Base_ptr->show();
28  }
```

Der2 also becomes an abstract class, not instantiable

Without Virtual destructor

```
3   class Base
4   {
5   public:
6       ~Base()
7       {  cout<<"Base class destr.\n";  }
8   };
9   class Derived: public Base
10  {
11  public:
12      ~Derived()
13      {  cout<<"Der. class destr.\n";  }
14  };
```

```
15  int main()
16  {
17      Base *base_ptr1, *base_ptr2;
18
19      base_ptr1 = new Base();
20      base_ptr2 = new Derived();
21
22      delete base_ptr1;
23      delete base_ptr2;
24  }
```

OUTPUT:

```
Base class destr.
Base class destr.
```

With Virtual destructor

```
3   class Base
4   {
5   public:
6       virtual ~Base()
7       {  cout<<"Base class destr.\n";  }
8   };
9   class Derived: public Base
10  {
11  public:
12      ~Derived()
13      {  cout<<"Der. class destr.\n";  }
14  };
```

```
15  int main()
16  {
17      Base *base_ptr1, *base_ptr2;
18
19      base_ptr1 = new Base();
20      base_ptr2 = new Derived();
21
22      delete base_ptr1;
23      delete base_ptr2;
24  }
```

OUTPUT:

```
Base class destr.
Der. class destr.
Base class destr.
```

base class reference without virtual function

```
3   class base
4   {
5   public:
6       void test()
7       { cout<<"base's test().\n"; }
8   };
9   class derived1 : public base
10  {
11  public:
12      void test()
13      { cout<<"derived1's test().\n"; }
14  };
15  class derived2 : public base
16  {
17  public:
18      void test()
19      { cout<<"derived2's test().\n"; }
20  };
```

```
21  int main()
22  {
23      derived1 d1;
24      derived2 d2;
25      base &b1 = d1;
26      base &b2 = d2;
27      b1.test();
28      b2.test();
29  }
```

OUTPUT

```
base's test().
base's test().
```

base class reference with virtual function

```
3  class base
4  {
5  public:
6      virtual void test()
7      { cout<<"base's test().\n"; }
8  };
9  class derived1 : public base
10 {
11 public:
12     void test()
13     { cout<<"derived1's test().\n"; }
14 };
15 class derived2 : public base
16 {
17 public:
18     void test()
19     { cout<<"derived2's test().\n"; }
20 };
```

```
21 int main()
22 {
23     derived1 d1;
24     derived2 d2;
25     base &b1 = d1;
26     base &b2 = d2;
27     b1.test();
28     b2.test();
29 }
```

OUTPUT

```
derived1's test()
derived2's test()
```

Some topics on Pointers…

Pointers and structures

Consider the following structure

```
struct inventory
{
    char  name[30];
    int  number;
    float  price;
} product[2], *p;
```

p=product; assigns the address of the zeroth element of **product** to **p**

or **p** points to **product[0];**

Pointers and structures (Contd…)

Its members are accessed using the following notation

p->name

p->number

p->price

The symbol **->** is called **arrow operator** (also known as **member selection operator**)

The member number can also be accessed using

(*p).number

Parantheses is required because **‘.’** has higher precedence than the operator *

Program to illustrate the use of structure pointers

```
3  struct invent
4  {
5      char name[30];
6      int number;
7      float price;
8  };
9  int main()
10 {
11     struct invent product[3], *ptr;
12     for( ptr = product; ptr < product+3; ptr++ )
13         cin>>ptr->name>>ptr->number>>ptr->price;
14     ptr=product;
15     while(ptr<product+3)
16     {
17         cout<<"\n"<<ptr->name<<"\t"<<ptr->number<<"\t"<<ptr->price;
18         ptr++;
19     }
20     return 0;
21 }
```

CONSTANT POINTERS..

1. **Pointer to constant**
2. **Constant pointer, non-constant data**
3. **Constant pointer, constant data**

1. Pointer to constant

const datatype * ptr;

- The value pointed by the pointer cannot be changed, but the pointer can be changed.

```
 3  int main()
 4  {
 5      int x = 10 , y = 20;
 6      const int *ptr = &x; // pointer to constant integer
 7      cout<< " *ptr = " << *ptr;
 8      // *ptr = 15 ; Error !
 9      ptr = &y;
10      cout<< "\n *ptr = " << *ptr;
11  }
```

OUTPUT

```
*ptr = 10
*ptr = 20
```

2. Constant pointer, non-constant data

int * const ptr;

- It creates a constant pointer to a non-constant integer.
- The value pointed by the pointer can be changed, but the pointer cannot be changed.

```
 3  int main()
 4  {
 5      int x = 10 , y = 20;
 6      int * const ptr = &x; // constant pointer to non-constant integer
 7      cout<< " *ptr = " << *ptr;
 8      *ptr = 15 ; // OK
 9      // ptr = &y; Error
10      cout<< "\n *ptr = " << *ptr;
11  }
```

OUTPUT

```
*ptr = 10
*ptr = 15
```

3. Constant pointer, constant data

const int * const ptr;

- It creates a constant pointer to a constant integer.
- The value pointed by the pointer cannot be changed, also the pointer cannot be changed.

```
 3  int main()
 4  {
 5      int x = 10 , y = 20;
 6      const int * const ptr = &x; // constant pointer to constant integer
 7      cout<<" *ptr = "<< *ptr;
 8      //*ptr = 15; // Error
 9      //ptr = &y; //Error
10      cout<<"\n *ptr = "<< *ptr;
11  }
```

Question-1

```
3   int main()
4   {
5       const int i = 11;
6       int *p;
7       p = &i;
8       *p = 15;
9   }
```

Error !

Question-2

```
3   int main()
4   {
5       const int i = 11;
6       int *p;
7       p = &i;
8   }
```

Error !

Solution:

```
3   int main()
4   {
5       const int i = 11;
6       const int *p;
7       p = &i;
8   }
```

Question-3

```
3   int main()
4   {
5       const int i = 11;
6       const int *p;
7       p = &i;
8       int &r = i;   Error !
9   }
```

Solution:

```
3   int main()
4   {
5       const int i = 11;
6       const int *p;
7       p = &i;
8       const int &r = i;
9   }
```