Pointer & Virtual Functions

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

We already discussed Compile Time Polymorphism. Now we learn Run Time Polymorphism.

When the selection of appropriate function or operator at Run Time , is called Late Binding or Dynamic Binding. Also known as Run Time Polymorphism.

Dynamic binding is one of the powerful features in C++. To achieve Dynamic binding we use pointer & virtual function.

Pointer:

A **pointer** is a variable whose value is the address of another variable. Since Pointer is also a kind of variable, thus pointer itself will be stored at different memory location.

Declaration: data_type *var-name;

Ex:
```
int *ip; // pointer to an integer .
double *dp; // pointer to a double .
```

Initialization:

Ex:
```
int *ip; int a;
 a=&ip;
```

Program:

```
int main ()
{
 int a, b;
int * ptr;
ptr= &a;
*ptr= 10;
ptr= &b;
*ptr= 20;
 cout << “Value of a is " << a<< '\n';
cout << “Value of b is " << b<< '\n';
return 0;
}
```

```
Output:    Value of a is 10
            Value of b is 20
```

C++ pointers vs. arrays

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing.

Example:

```
        const int MAX = 3;
         int main ()
        {
             int var[MAX] = {10, 100, 200};
             int *ptr;
             ptr = var;
        for (int i = 0; i < MAX; i++)
        {
        cout << "Address of var[" << i << "] = "; cout << ptr << endl;
        cout << "Value of var[" << i << "] = "; cout << *ptr << endl;
         ptr++;
        }
        return 0;
        }
```

Array of Pointers:

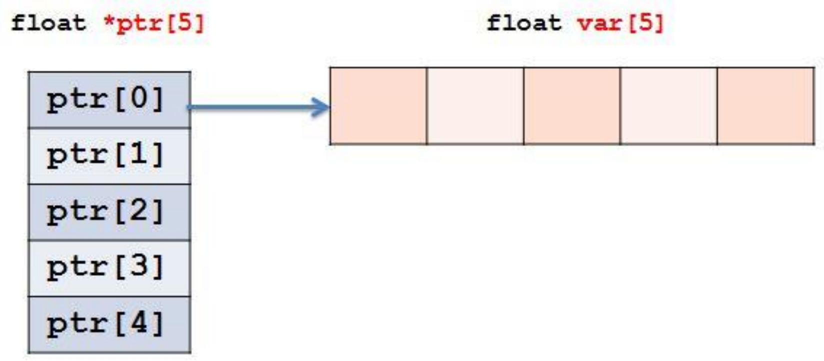
An array of pointers means an array of data items. Data items can be accessed either directly or by dereferencing the elements of pointer array.

Declaration: int * a[10];

This statement declares an array of 10 pointers, each of which points to an integer.

Program:

```
int main()
{
float var [5] = {1.1f , 2.2f , 3.3f , 4.4f , 5.5f};
float (*ptr)[5];

ptr =&var;
Cout<<"Value inside ptr\t ", ptr);

ptr =ptr+1;
Cout<<"Value inside ptr1\t ", ptr);

ptr =ptr+2;
Cout<<"Value inside ptr2\t ", ptr);

ptr =ptr+3;
Cout<<"Value inside ptr3\t ", ptr);
}
```



```
Output:
Value inside ptr : 2686696
Value inside ptr 1: 2686716
Value inside ptr 2: 2686756
Value inside ptr 3: 2686816
```

Pointers and Strings:

A string is one dimensional array of characters , which starts with the index 0 and ends with the null character '\0' in C++.

A pointer variable can access a string by referencing to its first character.

There are two ways to assign a value to a string:-

1) Use the character array. e.g. Char a[] = "hello";
2) Variable of type char*. e.g. char *a = "hello";

Program:

```
int main ()
 {
 char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
cout << "Greeting message: ";
cout << greeting << endl;
 return 0;
}
```

```
Output:   Hello
```

```
int main()
{
 char * s = "Joe";
 cout << s << endl;
 s = "Frederick";
 cout << s << endl;
 return 0;
}
```

```
Output:   Joe
          Frederick
```

Pointers to Functions:

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. It is also known as callback function.

Using function pointer user can select a function dynamically at run time. We can also pass a function as an argument to another function . Here the function is passed as a pointer. The function pointers cannot be referenced.

Like other function, we can declare a function pointer in C++.

Syntax : data_type(*function_name) ();

Example : int (*num(int x));

NOTE :

Remember that declaring a pointer only creates a pointer. It does not create actual function.

For this, we must define the task, which is to be performed by the function. The function must have the same return type & arguments.

Program:

```
void (*fun)(int, int);
void add (int i, int j)
{
cout<< “sum is :\t”<<i+j <<endl;
}

void subtract (int i , int j)
{
cout<<“subtract is:\t”<< i-j<<endl;
}

void main()
{
fun ptr;
ptr = &add;
ptr(1,2);
cout<< endl;
ptr = &subtract;
ptr(3,2);
getch();
}
```

```
Output:
Sum is : 3
Subtract is: 1
```

Pointers to Objects:

A pointer can point to an object created by a class. Such type of pointer is called Object Pointer.

Ex:
```
item a;
item *ptr;
ptr= &a;
```

Use of Object Pointer:

1) They are useful to create objects at run time.
2) We can also use an object pointer to access the public members of an object.

We can access member function in two ways:

1) When we use object then we use(.) dot operator to access the member functions.

2) But when we use object pointer then we use(*) arrow operator to access the member functions.

Ex: a.getdata(100,200);

ptr-> getdata(100,200);

OR

(*ptr).getdata(100,200); {Since *ptr is an alias of a,so we can also write this way)

We can also create an array of 10 objects using pointers.

Ex: item *ptr = new item[10];

Program:

```
class item
{
int code;
float price;
public:
void getdata(int a, float b)
{
code = a;
price =b;
}

void show()
{
cout<<“code:<<code<<“\n”<<“price”
<<price;
}
};
```

```
int main()
{
item *p = new item[2];
item *d = p;
int i,x;
float y;

for(i=0;i<2; i++)
{
cout<<“enter code & price “<<i+1;
cin>>x>>y;
p-> getdata(x,y);
p++;
}
for(i=0;i<2;i++)
{
cout<<“item”<<i+1<<“\n”;
d->show();
d++;
}
return 0;
}
```

This Pointers:

'this' pointer is a constant pointer that holds the memory address of the current object. It is not available for static member functions and friend functions.

Because static member functions can be called without any object and Friend functions are not members of a class. Only member functions have a this point .

Following are the situations , where we need 'this' pointer:

1) When local variable's name is same as member's name
2) To return reference to the calling object

We can access member function in two ways:

1) When we use object then we use(.) dot operator to access the member functions.

2) But when we use object pointer then we use(*) arrow operator to access the member functions.

Ex: a.getdata(100,200);

ptr-> getdata(100,200);

OR

(*ptr).getdata(100,200); {Since *ptr is an alias of a,so we can also write this way)

We can also create an array of 10 objects using pointers.

Ex: item *ptr = new item[10];

Program: 1) When local variable’s name is same as member’s name

```
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
         this->x = x;
   }
void print()
{
 cout << "x = " << x << endl;
}
};
```

```
int main()
{
   Test obj;
   int x = 20;
   obj . setX (x);
   obj . print();
   return 0;
}
```

```
Output:

x=20
```

Program: 1) To return reference to the calling object

```
Test& Test:: func ()
{
   return *this;
}

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0)
 {
 this->x = x; this->y = y;
 }
  Test & setX(int a)
 {
 x = a; return *this;
}
Test & setY (int b)
 {
 y = b; return *this;
 }
  void print()
 {
cout << "x = " << x << " y = " << y << endl;
 }
};

int main()
{
  Test obj1(5, 5);
   obj1.setX(10).  setY (20);

  obj1.print();
}
```

```
Output:

x=10
y=20
```

Pointers to Derived Classes:

We can use pointer not only to the base objects, but also to the objects of derived class . Pointers to objects of a base class are type compatible with pointers to objects of a derived class.

Ex: if B is a base class sand D is a derived class from B then a pointer declared as a pointer to B can also be a pointer to D..

See the following declarations:

```
B * bptr ;          // pointer to class B type variable
B b;                 // base object
D d;                // derived object
bptr = &b;         // cptr points to object b
```

We can make cptr to point to D also

```
bptr=&d;          // cptr points to d
```

Note:

1) There is a problem arise using bptr to access the public members of the derived class D. Using bptr, we can access only those members which are inherited from B and not the members that originally belong to D.

2) C++ permits a base pointer to point any object derived from that base, the pointer cannot be directly used to access to all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

Program

```
Class A
{
Public:
Int b;
Void show();
{
Cout<<“b=“<<b<<“\n”;
}
};
Class B : public A
{
Public:
Int d;
Void show();
{
Cout<<“b=“<<b<<“\n”<<“d
      =“<<d<<“\n”;
}
};
```

```
Int main()
{
A *ptr;
A obj;
Ptr = &obj;

Ptr-> b =100;
Ptr-> show();

B obj1;
Ptr = &obj1;
Ptr->b = 200;
Ptr->show();

B *ptr1;
Ptr = &obj1;
Ptr->d = 300;
Ptr->show();
}
```

```
Output:

b=100
B=200
B=200
D=300
```

Virtual Function in C++

If there are member function with same name base class and derived class, virtual function gives programmer the capability to call member function of different class by a same function call depending upon different context.

When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the virtual function.

We can define Virtual function as a function in base class, which is override in the derived class, and which tells the compiler to perform Late Binding on this function .

Virtual Keyword is used to make a member function of the base class Virtual .

Late Binding:

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called Dynamic Binding or Runtime Binding.

Problem without Virtual keyword:

If a base class & derived class has some function and if you write code to that function using pointer of base class, then the function in base class executed even if the object of derived class is referred with that pointer variable.

See the next example.

Program: 1) Problem without Virtual keyword:

```
class Base
{
 public:
 void show()
{
 cout << “Hello";
}
};

class Derived :  public Base
{
 public:
void show()
 {
 cout << “Hi";
 }
 };
```

```
int main()
{
 Base* b;
 Derived d;

  b = &d;
  b->show();    //    Early binding occurs
}
```

This program shows, even if the object of derived class is put in the pointer to base class, show() of base class executed.

Output: Hello

If you want to execute the member function of derived class then you can declare show() in base class virtual, which makes that function existing in appearance only but you can't call that function.

See the next example.

Program: 2) Using Virtual keyword:

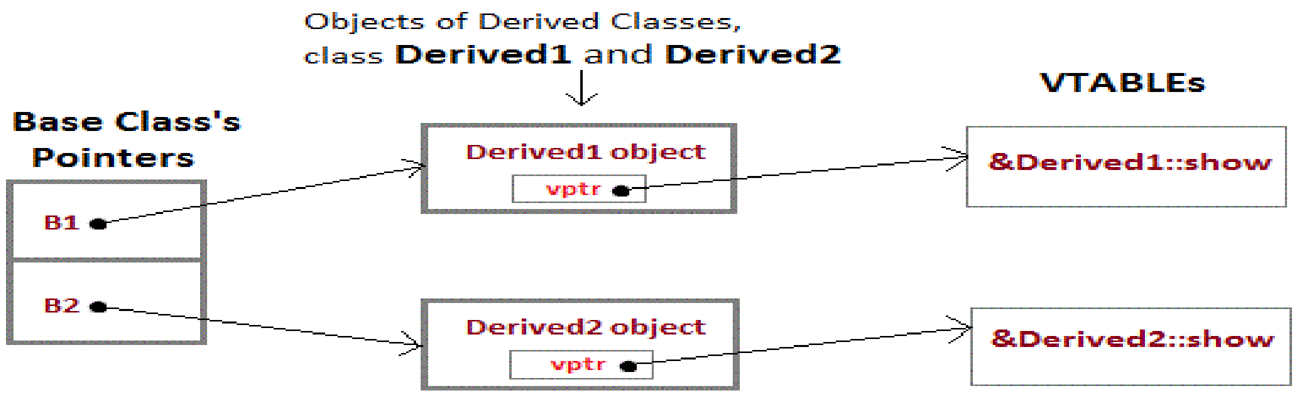
```
class Base
{
 public:
virtual  void show()
{
 cout << “Hello";
}
};

class Derived :  public Base
{
 public:
void show()
 {
 cout << “Hi";
 }
 };
```

```
int main()
{
 Base* b;
 Base  c;
 Derived d;

  b=&c;
  b->show();

  b
  b
}
```

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

```
 Output:    Hello
            Hi
```

Mechanism of Late Binding:



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called pointer, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the pointer.

Important Points to Remember :

1) Only the Base class Method's declaration needs the Virtual Keyword, not the definition.
2) If a function is declared as virtual in the base class, it will be virtual in all its derived classes.
3) The address of the virtual Function is placed in the VTABLE and the compiler uses VPTR(vpointer) to point to the Virtual Function.

Rules for Virtual Functions:

1. Only non static member functions can be *virtual*.
2. The virtual characteristic is inherited. Thus, the derived class function is automatically virtual, and the presence of the virtual keyword is usually omitted.
3. Constructors cannot be virtual.
4. Destructors can be virtual. As a rule of thumb, any class having virtual functions should have a virtual destructor.

Pure Virtual Function in C++

Pure Virtual Functions are the virtual functions with no definition. They start with virtual keyword and ends with 0.

Syntax: virtual void f() = 0;

Abstract class contain the pure virtual functions.

The main objective of an abstract base class is to provide some triats t the derived classes and to create a base pointer required for achievigng run time polymorphism.

Program:

```
class B     //Abstract base class
{
 public:
 virtual void show() = 0;          //Pure Virtual Function
};

class D:public B
{
public:
 void show ()
 {
 cout << "Implementation of Virtual Function in Derived
      class";
}
};
```

```
int main()
{
 B obj;     //Compile Time Error
 B *b;
 D d;
  b = &d;
  b->show();
}
```

```
Output:
Implementation  of  Virtual
Function  in  Derived  class
```