# Virtual Function, Run Time Polymorphism and miscellaneous C++ features

## Pointers

One important use for pointers is in the dynamic allocation of memory, carried out in C++ with the keyword new and delete.

### New and Delete Operator

Pointer provides the necessary support for C++ powerful dynamic memory allocation system. Dynamic allocation is the means by which a program can obtain memory while it is running. C uses malloc( ) and calloc( ) functions to allocate memory dynamically at run time. Similarly, it uses the function free( ) to free dynamically allocated memory.

Although C++ supports these functions, it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in a better and easier way. An object can be created by using new and destroyed by using delete as and when required.

new operator is used to allocate memory dynamically i.e., at run time. new operator obtains memory from the operating system and returns a pointer to the starting point.

**The syntax for the *new* operator:**

pointer\_variable=new data-type;

Where pointer\_variable is a previously declared pointer of type type\_name. type\_name can be any basic data type or user-defined object (enum, class, and struct included).

**Example:**

int \*p;

//int \*p = new int;

p = new int;

float \*q;

//int \*q = new float;

q = new float;

Type of variable mentioned on the left hand side and the type mentioned on the right hand side should match.

When a data object is no longer needed, it should be destroyed to release the memory space for reuse. For this purpose, we use ***delete*** unary operator.

**The general syntax for delete operator:**

delete pointer\_variable;

The pointer\_variable is the pointer that points to a data object created with new.

#### Example:

delete p;

delete q;

## Pointer to Objects

There are two ways to declare pointer to objects.

* **Usual way:** Just like pointers to normal variables, we can have pointers to class variables, i.e. objects.

**Syntax**:

class\_name \* ptr\_name = & object\_name;

**Example:**

A b;

A \*a = & b; **// Here, b is an object and a is pointer to it.**

* **Using new Operator: new operator can be used to define pointer to objects.**

**Syntax:**

class\_name \*ptr\_name = new class\_name;

**Example:**

A \*a = new A; **// Here, a is pointer to unnamed object.**

## Pointer to Derived Class

Base class object pointer can point any type of derived class objects. Pointers to objects of a base class are type compatible with pointers to objects of derived class.

Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if B is a base class and D is a derived class from B, then a pointer declared as a pointer to B can also be a pointer to D.

Consider the following declaration:

B\*cptr; **// pointer to class B type variable**

B b ; **// base object**

D d ; **// derived object**

cptr=&b ; **// cptr points to object b**

We can make cptr to point to the object d as follows:

cptr=&d ; // cptr points to object d

## Accessing Class Member Using Pointer to Objects

We can access class members i.e., member functions and data members using pointer to object. This can be done by using object pointer and arrow operator(->).

**Syntax:**

**object\_pointer->class\_member;**

**Example:**

class Sample

{

int a, b;

public:

int p;

Sample(int x, int y )

{

a=x;

b=y;

}

void display( )

{

cout<< " a, b, p :"<<a<<b<<p;

}

};

int main( )

{

Sample s(7,8);

s.p=9;

s.display();

Sample \*a =&s;

a->p=10; **//accessing public data member p using object pointer a.**

a->display(); **//accessing public member function display() using object pointer a.**

**//Another way to define pointer to object.**

Sample \* k = new Sample; **//using new operator, here object pointer k points to unnamed object.**

k->p=80;

k->display();

return 0;

}

## this Pointer in C++

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

this is a special pointer which points to the object that is currently invoking a particular member function. For example, the function call a.display() will set the pointer “this” to the address of the calling object a, and again b.display() will set the pointer "this" to the address of the object b. this pointer is automatically passed to a member function when it is called. Therefore, inside a member function, this pointer is used to refer to the invoking object.

**It can be used to access the data in the object. Also, this pointer can be used to find out the address of the calling object, to return values.**

**Example:**

#include <iostream>

using namespace std;

class thisimp

{

private:

int main( )

{

emp e1, e2;

e1.setdata(10);

e2.setdata(30);

e1.showdata( ) ;

e2.showdata( ) ;

}

int i;

public:

void setdata (int num)

{

i = num; //one way to assign data.

this ->i = num; //another way to assign data, remember "this" is a pointer to the calling object.

}

void showdata( )

{

cout<< "i:" <<i<<endl ; //one way to display data

cout<< " address is:" <<this<<endl ;

cout<< " i:"<< this->i; //another way to display

}

} ;

## Polymorphism

Polymorphism is one of the crucial features of OOP. Polymorphism means one name, multiple forms.

**Classification of Polymorphism:**

1. Compile time polymorphism

2. Run time polymorphism

### Compile Time Polymorphism

Also called **early binding** or **static binding** or **static linking**. In compile time polymorphism compiler selects the appropriate member function for particular function call at the compile time. Member functions are selected for invoking at the compile time by matching arguments, both type and number. The information regarding which function to invoke that matches a particular call is known in advance during compilation. Function overloading, constructor overloading, operator overloading, all these are examples of compile time polymorphism.

**Example:**

#include <iostream>

void area(float r) ;

void area(float l, float b) ;

void main( )

{

float r1, l1, b1 ;

cout<< “enter value of r1:” ; cin>>r1 ;

cout<< “enter value of l1:” ; cin>>l1 ;

cout<< “enter value of b1:” ; cin>>b1 ;

cout<< “Area of circle is”<<endl; area(r1) ;

cout<< “Area of rectangle is”<<endl ; area(l1, b1) ;

}

void area (float r)

{

float a=3.14\*r\*r ;

cout<< “Area=”<<a<<endl ;

}

void area(float l , float b)

{

float a1=l\*b ;

cout<< “Area=”<<a1<<endl ;

}

In above example two functions have the same name, but question is how compile differentiates these two. Actually, C++ compiler differentiates these two by argument. In one area function, there is only one argument which is float type but in second area function there are two arguments both are float type. If any program has two functions both have same name and number of argument same, then compiler differentiates these by type of arguments.

### Run Time Polymorphism

In **run time polymorphism the** appropriate member function is selected while program is running. It is also called **late binding** or **dynamic binding** because the appropriate function is selected dynamically at run time. This type of binding requires virtual functions and base class pointers.

### Virtual Functions

A virtual function a member function which is declared within base class and is re-defined (Overridden) by derived class. A virtual function is a function in a base class that is declared using the keyword virtual. A virtual function is a member function in base class, which is overridden in the derived class (i.e., redefined), and which tells the compiler to perform late binding on this function.

When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. A function can be made virtual by placing the keyword virtual before its normal declaration. When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class’s version of the function.

Virtual Functions are used to support "Run time Polymorphism". ... When the virtual function is called by using a Base Class Pointer, the Compiler decides at Runtime which version of the function i.e. Base Class version or the overridden Derived Class version is to be called. This is called Run time Polymorphism.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call. The resolving of function call is done at Run-time.

Virtual functions are useful when we have number of objects of different classes but want to put them all on a single list and perform operation on them using same function call.

When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

class base\_class

{

…….

……

public:

virtual return\_type function\_name(arguments)

{

**//function body**

}   
};

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) an Early binding(Compile time) is done according to the type of pointer.

*To achieve run time polymorphism, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes(i.e., it requires function overriding). The function in the base class is declared as virtual using the keyword virtual. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.*

**Some Rules for Virtual functions:**

* The virtual functions must be members of some class. And they must be defined.
* They cannot be static members.
* They are accessed by using object pointers.
* A virtual function can be a friend of another class.
* We cannot have virtual constructors, but we can have virtual destructor.
* The prototypes of the virtual in the base class and the corresponding member function in the derived class must be same. If not same, then C++ treats them as overloaded functions (having same name, different arguments) thereby the virtual function mechanism is ignored.
* The base pointer can point to any type of the derived object, but vice-versa is not true i.e. the pointer to derived class object cannot be used to point the base class object.

## Normal Member (*Non-virtual*) Functions Accessed with Base Class Pointers

When base class and derived classes all have functions with the same name, and we want to access these derived class functions with single base class object pointer without using virtual function.

**Example:**

class Base

{

public:

void display()

{

cout<<" This is base "<<endl;

}

};

class Derv1: public Base

{

public:

void display()

{

cout<<" this is derived 1 ";

}

};

class Derv2: public Base

{

public:

void display()

{

cout<<" This is derived2 ";

}

};

int main()

{

Base \* ptr; **//pointer to base class declaration.**

Derv1 D1; **//object of derived class 1.**

Derv2 D2; **//object of derived class 2.**

ptr =&D1; **//ptr points to derived class object D1.**

ptr ->display(); **//calling display() function.**

ptr =&D2; **//ptr now points to derived class object D1.**

ptr ->display(); **//again calling display() function.**

}

**OUTPUT:**

This is base

This is base

As we can see, the function display() in the base class is always executed. The compiler ignores the contents of pointer ***ptr*** and always chooses the member function that matches the type of the pointer, in this case Base. So it does not implement polymorphism i.e., single name multiple form.

The reason for the this output is that the call of the function display() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the display() function is set during the compilation of the program.

## Virtual Member Functions Accessed with Base Class Pointers

Let’s make a single change in the above program. We will place the keyword **virtual** in the declaration for the **display()** function in the base class.

**Consider the following simple program which is an example of runtime polymorphism.**

The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

**Run time polymorphism: Example 0**

#include <iostream>

using namespace std;

class Base

{

public:

virtual void display() **//virtual function.**

{

cout<<" This is base ";

}

};

class Derv1: public Base

{

public:

void display()

{

cout<<" this is derived 1 "<<endl;

}

};

class Derv2: public Base

{

public:

void display()

{

cout<<" This is derived2 ";

}

};

int main()

{

Base \*ptr; **//base class pointer declaration**.

Derv1 D1; **//object of derived class 1.**

Derv2 D2; **//object of derived class 2.**

ptr=&D1; //ptr points to derived class object D1.

ptr->display(); **//calling display() function, invokes member function of derived1. Why?**

ptr=&D2; //ptr now points to derived class object D1.

ptr->display(); **//again calling display() function, now invokes member function of derived2.** **Why?**

}

**OUTPUT:**

This is derived 1

This is derived 2

Now, the member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of Derv1 to that of Derv2, and the particular instance of display() that is executed also changes. So the same function call,

ptr->display();

executes different functions, depending on the contents of ptr(i.e., object pointed by ptr).

Here, the compiler does not know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class. Which version of display() does the compiler call? In fact the compiler does not know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointing to by ptr, the appropriate version of display() will be called, exhibiting late binding.

#### Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

// CPP program to illustrate

// concept of Virtual Functions

#include<iostream>

using namespace std;

class base

{

public:

virtual void print ()

{ cout<< "print base class" <<endl; }

void show ()

{ cout<< "show base class" <<endl; }

};

class derived:public base

{

public:

void print ()

{ cout<< "print derived class" <<endl; }

void show ()

{ cout<< "show derived class" <<endl; }

};

int main()

{

base \*bptr;

derived d;

bptr = &d;

bptr->print(); //virtual function, binded at runtime

bptr->show(); // Non-virtual function, binded at compile time

}

**Explanation:** Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer ‘bptr’ contains the address of object ‘d’ of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) an Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be binded at run-time (output is print derived class as pointer is pointing to object of derived class ) and show() is non-virtual so it will be binded during compile time(output is show base class as pointer is of base type ).

**Run time polymorphism: Example1**

#include<iostream>

class tape: public media

{

int length;

public:

tape(string a, float b, int p): media(a, b)

{

length=p;

}

void show()

{

cout<<" Title: "<<title<<endl;

cout<<" Price: "<<price<<endl;

cout<<" Length: "<<length;

}

};

int main()

{

media \*p;

tape t(" Rihana",420.23,45);

book b("Othelo",1080.23,456);

p=&t;

p->show();

p=&b;

p->show();

}

#include<string>

using namespace std;

class media

{

protected:

string title;

float price;

public:

media(string a, float b)

{

title=a;

price=b;

}

virtual void show(){ } **//Empty virtual function.**

};

class book: public media

{

int pages;

public:

book(string a, float b, int p): media(a, b)

{

pages=p;

}

void show()

{

cout<<" Title: "<<title<<endl;

cout<<" Price: "<<price<<endl;

cout<<" Pages: "<<pages;

}

};

**Run time polymorphism: Example2**

class Shape

{

protected:

int length;

int breadth;

public:

Shape(int l, int b)

{

length=l;

breadth=b;

}

void display()

{

cout<<"Length: "<<length<<"\t"<<"Breadth: "<<breadth<<endl;

}

virtual void area()

{

int main()

{

Shape\*s; **//base class pointer declaration.**

Rectangle R(7, 8);

Square S(7, 7);

R.display();

S.display();

s=&R; //s points to derived class object R.

s->area(); //invokes area() function of derived class **Rectangle**.

s=&S; //s now points to derived class object S.

s->area(); //invokes area() function of derived class **Square**.

return 0;

}

**//empty body.**

}

};

class Rectangle: public Shape **//derived from base class shape**

{

public:

Rectangle(int a, int b): Shape(a, b)

{

}

void area()

{

cout<<"Area of rectangle: "<<length\*breadth<<endl;

}

};

class Square: public Shape **//derived from base class shape**

{

public:

Square(int a, int b): Shape(a, b)

{

}

void area()

{

cout<<"Area of square: "<<length\*breadth;

}

};

## Pure Virtual Functions

A pure virtual function (or abstract function) in C++ is a virtual function for which don’t have implementation (body), we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

class Test  **// An abstract class, since contains pure virtual function**

{

// Data members of class

public:

// Pure Virtual Function

virtual void show() = 0;

/\* Other members \*/

};

A pure virtual function is a virtual function that has no definition within the base class (i.e. with no function body). Pure virtual function is one with the expression = 0 added to the declaration (a virtual function equated to 0). Once we have virtual function in base class, then it must be redefined in every derived class.

**Example:**

class My

{

……………

…………….

public:

virtual void show()=0; **//Pure virtual function.**

};

**If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.**

The following example demonstrates the same.

#include<iostream>

using namespace std;

class Base

{

public:

virtual void show() = 0;

};

class Derived : public Base { };

int main(void)

{

Derived d; //error, since class Derived does not have function named show() overridden so

return 0; also an abstract class.

}

## Abstract Base class

A base class that has at least one pure virtual function is called abstract base class. It designed only to act as a base class. An abstract base class cannot be used to create objects. But we can define base class pointers. It provides only an interface for its derived classes. The main objective of an abstract base class is to provide some attributes to the derived class and to create a base class pointer required for run time polymorphism.

**Example:**

class A

{

protected:

int data;

public:

A(){}

A(int d)

class C : public A **//derived class C.**

{

public:

C(int d) : A(d)

{

**//empty body.**

}

void show()

{

cout<<data;

}

};

int main()

{

A \*a;

A z; **// error (Why?)**

B b(5);

C c(6);

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

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

}

{

data = d;

}

virtual void show() = 0; **//pure virtual function**

};

class B : public A **//derived class B.**

{

public:

B(int d) : A(d)

{

**//empty body.**

}

void show()

{

cout<<data<<endl;

}

};

**Another Example:**

class Shape

{

protected:

int length;

int breadth;

public:

Shape(int l, int b)

{

length=l;

breadth=b;

}

void display()

{

cout<<"Length: "<<length<<"\t"<<"Breadth: "<<breadth<<endl;

}

virtual void area()=0; **//pure virtual function.**

};

class Rectangle: public Shape **//derived from base class shape**

{

public:

Rectangle(int a, int b): Shape(a, b)

{

//empty body

}

void area()

{

cout<<"Area of rectangle: "<<

length\*breadth<<endl;

}

};

class Square: public Shape **//derived from base class shape.**

{

public:

Square(int a, int b): Shape(a, b)

{

//empty body

}

void area()

{

cout<<"Area of square: "<<length\*breadth;

}

};

int main()

{

Shape\*s[2]; **//array of base class pointer declaration.**

Rectangle R(7, 8);

Square S(7, 7);

R.display();

S.display();

s[0]=&R; **//s[0] points to derived class object R.**

s[0]->area(); **//invokes area() function of derived class Rectangle.**

s[1]=&S; **//s[1] points to derived class object S.**

s[1]->area(); **//invokes area() function of derived class Square.**

return 0;

}

## Abstract vs. Concrete Classes

Concrete means "existing in reality or in real experience; perceptible by the senses; real", whereas abstract means "Not applied or practical; theoretical.

A concrete class is a class that can be used to create an object. An abstract class cannot be used to create an object.

An abstract class is one that has one or more pure virtual function. Whereas a concrete class has no pure virtual functions. A base class can be either abstract or concrete and a derived class can also be either abstract or concrete.

**Example**

class Beta

{

………

………

void draw();

………

………

};

class Alpha class Alpha

{

………

………

virtual void draw()=0;

………

………

};

In above example, class Alpha is abstract class and class Beta is concrete class. Concrete classes do not have pure virtual functions.

## Early & Late Binding

The differences between early and late binding are given below:

|  |  |
| --- | --- |
| Early Binding | Late Binding |
| 1. It is also known as compile time polymorphism. It is called so because compiler selects the appropriate member function for particular function call at the compile time.  2. The information regarding which function to invoke that matches a particular call is known in advance during compilation. That is why it is also called as early binding.  3. The function call is linked with particular function at compiler time statically. So, it is also called static binding.  4. This type of binding can be achieved using function overloading and operator overloading. | 1. It is also known as run time polymorphism. It is called so because the appropriate member functions are selected while the program is executing or running.  2. The compiler doesn’t know which function to bind with particular function call until program is executed so it is also called late binding.  3. The selection of appropriate function is done dynamically at run time, so it is also called dynamic binding.  4. This type of binding is achieved using virtual function and base class pointer. |

NOTE: I have already explained and discussed about static function, friend class, assignment and copy initialization , copy constructor in our class.