

POINTER TO OBJECTS (OBJECT POINTER)

Er. Rudra Nepal

OBJECT POINTER

The pointer pointing to objects is referred to as an object pointer.

Declaration

`Class_name *object_pointer_name;`

`Eg. student *ptr;`

Here, ptr is an object pointer of the student class type that has been declared. where the student is already defined class.

Initialization

`object_pointer_name=&object;`

`Eg. ptr=&st;`

Here, ptr is an object pointer of student class type and st is an object of the class student.

Note: When accessing members of a class using an object pointer the arrow operator (->) is used instead to dot operator.

Example:

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
public:
void getdata()
{
cout<<"Enter student Name"<<endl;
cin>>name;
cout<<"Enter student Rollno"<<endl;
cin>>roll;
}
```

```
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Rollno:"<<roll<<endl;
}
};

int main()
{
student st;
student *ptr;
ptr=&st;
ptr->getdata();
ptr->display();
return 0;
}
```

CREATING OBJECTS AT RUNTIME USING OBJECT POINTERS

Creating objects at runtime using object pointers

Object pointers are also used to create the object at runtime by using it with new operator as follows:

Eg. item *ptr=new item;

This statement allocates enough memory space for the object and assigns the address of the memory space to ptr.

We can also create an array of objects using pointers. For example, the statement

item *ptr=new item[10];

Creates memory space for an array of 10 objects of item.

Example:

Program:

```
#include<iostream>
using namespace std;
class item
{
private:
int code;
float price;
public:
void getdata(int c,float p)
{
code=c;
price=p;
}
void display()
{
cout<<"Code="<<code<<endl;
cout<<"Price="<<price<<endl;
}
};
```

```
int main()
{
int n,i,x;
float y;
cout<<"Enter the number of item"<<endl;
cin>>n;
item *ptr=new item[n];
item *d=ptr; //&ptr[0]
for(i=0;i<n;i++)
{
cout<<"Input code and price of item"<<endl;
cin>>x>>y;
ptr->getdata(x,y);
ptr++;
}
for(i=0;i<n;i++)
{
cout<<"Item:"<<i+1<<endl;
d->display();
d++;
}
return 0;
}
```

POINTER TO DERIVED CLASS

Can you derive a pointer from a base class? Explain with a suitable example

Yes, we can derive a pointer from a base class. Pointers to object of base class are type compatible with pointer to objects of the 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 the derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**. consider the following declarations.

```
B *bptr; //pointer to class B type variable
```

```
B b; //base object
```

```
D d; //derived object
```

```
bptr=&b; //bptr points to object b
```

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

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

This is perfectly valid with C++ because d is an object derived from the class B.

POINTER TO DERIVED CLASS

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. we can access only those members which are inherited from B and not the members that originally belong to D. We may have to use another pointer declared as pointer to derived type.

Example:

```
#include<iostream>
using namespace std;
class B
{
public:
void display()
{
cout<<"Base class"<<endl;
}
};
class D:public B
{
public:
void display()
{
cout<<"Derived class"<<endl;
}
};
```

```
int main()
{
B b1;
B *bptr; //base pointer
bptr=&b1; //base address
cout<<"bptr points to base object"<<endl;
bptr->display();
D d1;
bptr=&d1; //address of the derived object
cout<<"bptr now points to derived object"<<endl;
bptr->display(); //access to base class member
D *dptr;
dptr=&d1;
cout<<"dptr is a derived type pointer"<<endl;
dptr->display();
cout<<"Using ((*D)bptr)"<<endl;
((D*)bptr)->display();
return 0;
}
```

THIS POINTER

- **this pointer stores the address of current calling object.**
- For example the function call `A.max()` will set the pointer `this` to the address of the object `A`.
- **The starting address is the same as the address of the first variable in the class structure.**
- **this pointer is automatically passed to a member function when it is called.**

The pointer `this` acts as an implicit argument to all member function.

- **this pointers are not accessible for static member functions.**
- **this pointers are not modifiable.**

1. THIS POINTER CAN BE USED TO REFER CURRENT CLASS INSTANCE VARIABLE.

Example:

```
#include<iostream>
using namespace std;
class Employee
{
private:
int eid;
float salary;
public:
Employee(int eid,float salary)
{
this->eid=eid;
this->salary=salary;
}
void display()
{
cout<<"Employee ID="<<eid<<endl;
cout<<"Salary="<<salary<<endl;
}
};
int main()
{
Employee e1(101,25452.55);
Employee e2(102,54485.25);
e1.display();
e2.display();
return 0;
}
```

2. ONE IMPORTANT APPLICATION OF THE POINTER THIS IS TO RETURN THE OBJECTS IT POINTS TO. FOR EXAMPLE, THE STATEMENT

RETURN *THIS;

INSIDE A MEMBER FUNCTION WILL RETURN THE OBJECT THAT INVOKED THE FUNCTION.

```
#include<iostream>
#include<string.h>
using namespace std;
class person
{
char name[20];
float age;
public:
person()
{
}
person (char n[ ],float a)
{
strcpy(name,n);
age=a;
}
person greater(person x)
{
if(x.age>=age)
{
return x;
}
else
{
return *this;
}
}
```

```
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Age:"<<age<<endl;
}
int main()
{
person p1("Ram",52);
person p2("Hari",24);
person p3;
p3=p1.greater(p2);
cout<<"Elder person is:"<<endl;
p3.display();
return 0;
}
```

VIRTUAL FUNCTIONS

- Virtual means existing in appearance but not in reality.
- Virtual function is declared by using a keyword 'virtual' preceding the normal declaration of a function.
- When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base pointer rather than the type of pointer.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects.
- When base class pointer contains the address of the derived class object, always executes the base class function. Here, the compiler simply ignores the contents of the (base) pointer and chooses the member function that matches the type of the pointer.
- This issue can only be resolved by using the 'virtual' function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of object pointed by the base class pointer. In this way runtime polymorphism can also be achieved.

```
#include<iostream>
using namespace std;
class B
{
public:
virtual void show()
{
cout<<"Show base"<<endl;
}
};
class D:public B
{
public:
void show()
{
cout<<"Show derived"<<endl;
}
};
```

```
int main()
{
B b1;
D d1;
B *bptr;
cout<<"bptr points to base"<<endl;
bptr=&b1;
bptr->show();//calls base class function
cout<<"bptr points to derived"<<endl;
bptr=&d1;
bptr->show();//calls derived class function
return 0;
}
```

RULES OF VIRTUAL FUNCTION

Rules of Virtual Function

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor. While a base pointer can point to any of the derived object, the reverse is not true. That is to say. We cannot use a pointer to derived class to access an object of base type.

1. WHAT IS VIRTUAL FUNCTION? WHEN AND HOW TO WE MAKE FUNCTION VIRTUAL? EXPLAIN WITH A SUITABLE EXAMPLE.

A virtual function is a member function declared in the base class with the keyword `virtual` and uses a single pointer to base class pointer to refer to objects of different classes.

When we use the same function name in both base and derived classes, the function in the base class is declared as `virtual` using the keyword `virtual` preceding its normal declaration.

When a function is made `virtual`, C++ determines which function is used at runtime 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 object of different versions of the `virtual` functions.

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. Furthermore, when we use `virtual` functions, different functions can be executed by the same function call. The information regarding which function to invoke is determined at run time.

Program:

```
#include<iostream>
using namespace std;
class B
{
public:
virtual void show()
{
cout<<"Show base"<<endl;
}
};
class D:public B
{
public:
void show()
{
cout<<"Show derived"<<endl;
}
};
```

```
int main()
```

```
{
```

```
B b1;
```

```
D d1;
```

```
B *bptr;
```

```
cout<<"bptr points to base"<<endl;
```

```
bptr=&b1;
```

```
bptr->show();//calls base class function
```

```
cout<<"bptr points to derived"<<endl;
```

```
bptr=&d1;
```

```
bptr->show();//calls derived class function
```

```
return 0;
```

```
}
```

Q. What happens when a base and derived classes have the same functions with the same name and these are accessed using pointers with and without using virtual functions.

```
#include<iostream>
using namespace std;
class B
{
public:
void display()
{
cout<<"Display base"<<endl;
}
virtual void show()
{
cout<<"Show base"<<endl;
}
};
```

```
class D:public B
{
public:
void display()
{
cout<<"Display derived"<<endl;
}
void show()
{
cout<<"Show derived"<<endl;
}
int main()
{
B b1;
D d1;
B *bptr;
cout<<"bptr points to base"<<endl;
bptr=&b1;
bptr->display();
//calls base class function
bptr->show();
//calls base class function
cout<<"bptr points to derived"<<endl;
bptr=&d1;
bptr->display();
//calls base class function
bptr->show();
//calls derived class function
return 0;
}
```

Output:

bptr points to the base

Display base

Show base

bptr points to derived

Display base

Show derived

When bptr is made to point the object d (ie. bptr=&d1),

the statement

bptr->display();

Calls only the function associated with the B.
(ie.B::display())

This is because the compiler actually ignores the content
of the pointer bptr and chooses a
member function that matches the type of the pointer.
whereas the statement

bptr->show();

calls the derived version of show(). This is because
function show() has been made virtual in
Base class.

This is because first the base pointer has the address of
the base class object, then its content is
changed to contain the address of the derived object.

3. How can you achieve runtime polymorphism in C++? Discuss with a suitable example.

We should use virtual functions and pointers to objects to achieve run-time polymorphism. For this, we use functions having same name, same number of parameters, and similar types of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword `virtual`. A virtual function uses a single pointer to base class pointer to refer to all the derived objects.

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

```
#include<iostream>
using namespace std;
class base
{
public:
    virtual void show()
    {
        cout<<"Base Class Show function"<<endl;
    }
};

class derived:public base
{
public:
    void show()
    {
        cout<<"Derived Class Show function "<<endl;
    }
};

int main()
{
    base b1,*bptr;
    derived d1;
    bptr=&b1;
    bptr->show();
    bptr=&d1;
    bptr->show();
    return 0;
}
```

A bookshop sells both books and videotapes. Create an abstract class known as media that stores the title and price of a publication. Create two child classes one for storing the number of pages in a book and another for storing the playing time of a tape. A function display is used in all the classes to display the class contents. Create necessary constructors in the child classes to store the information. In the main display the information regarding the book and tape using the base pointer (an object pointer of the class media)

```
#include<iostream>
#include<string.h>
using namespace std;
class media
{
protected:
char title[20];
float price;
public:
media(char t[ ] , float p)
{
strcpy(title,t);
price = p;
}
virtual void display() = 0;
};
class book : public media
{
int pages;
public:
book(char t[], float p, int pag):media(t,p)
{
pages = pag;
}
```

```
void display()
{
cout<<"Title:"<<title<<endl;
cout<<"Price:"<<price<<endl;
cout<<"Pages:"<<pages<<endl;
}
};

class tape:public media
{
int time;
public:
tape(char t[ ], float p, int tm):media(t,p)
{
time = tm;
}
void display( )
{
cout <<"Title:"<<title<<endl;
cout <<"Price:"<<price<<endl;
cout <<"play time(mins):"<<time<<endl;
}
};
```

```
int main()
{
media *m[2];
book b("OOP",550.25,350);
tape t("computing concepts",255.6,55);
m[0] = &b;
cout<<"Information of Book:"<<endl;
m[0]->display();
cout<<"Information of media:"<<endl;
m[1] = &t;
m[1]->display();
return 0;
}
```

PURE VIRTUAL FUNCTION (DEFERRED METHODS/ABSTRACT METHODS)

- Pure virtual function (Deferred methods/Abstract methods)
- A virtual function will become a pure virtual function when we append "=0" at the end of the declaration of the virtual function.
- Example: `virtual void display() =0;`
- Pure virtual functions are also known as “do-nothing” functions.
- A pure virtual function is a function declared in a base class that has no definition (implementation/body).
- It serves only as a placeholder.
- The child classes are allowed to inherit them.
- In this situation, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Otherwise, the compilation error will occur.

Program:

```
#include<iostream>
using namespace std;
class Book
{
public:
virtual void display()=0;
};
class Math:public Book
{
public:
void display()
{
cout<<"We are studying Math "<<endl;
}
};
```

```
class OOP:public Book
{
public:
void display()
{
cout<<"We are studying OOP"<<endl;
}
};
int main()
{
Book *bptr;
Math m;
OOP o;
bptr=&m;
bptr->display();
bptr=&o;
bptr->display();
return 0;
}
```

ABSTRACT CLASS

- A class having at least one pure virtual function is called an abstract class.
- The object of abstract classes cannot be created.
- Pointers to abstract classes can be created for selecting the proper virtual function.
- An abstract class is designed to act only as a base class. It is a design concept in program development and provides a base upon which the program is built.
- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. For example, let Shape be a base class. We cannot provide an implementation of function draw() in Shape, but we know every derived class must have the implementation of draw(). In this case, concept of abstract class is used.

Example:

```
#include<iostream>
using namespace std;
class shape
{
public:
virtual void draw()=0;
};
class square:public shape
{
public:
void draw()
{
cout<<"Implementing method to draw square"<<endl;
}
};
```

```
class circle: public shape
```

```
{
```

```
public:
```

```
void draw()
```

```
{
```

```
cout<<"Implementing a method to draw a circle"<<endl;
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
shape *bptr;
```

```
square s;
```

```
circle c;
```

```
bptr=&s;
```

```
bptr->draw();
```

```
bptr=&c;
```

```
bptr->draw();
```

```
return 0;
```

```
}
```

FUNCTION OVERRIDING (OVERRIDING BASE CLASS MEMBERS)

- Function overriding (Overriding base class members)
- If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding.

Requirements for Overriding a Function

- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same declaration in both base and derived class, which means the same name, same return type, and same parameter list.

Example:

```
#include<iostream>
using namespace std;
class A
{
public:
void display()
{
cout<<"This is Base class"<<endl;
}
};

class B : public A
{
public:
void display()
{
cout<<"This is Derived class"<<endl;
}
};
```

```
int main()
{
B b;
b.display();
return 0;
}
```

Here, the function display() is overridden.

If the function is invoked from an object of the derived class, then the function in the derived is executed.

If the function is invoked from an object of the base class, then the base class member function is invoked.

VIRTUAL DESTRUCTORS

- Constructors cannot be virtual due to the following reasons.
- To create an object the constructor of the object class must be of the same type as the class. But it is not possible with the virtually implemented constructor.
- At the time of calling the constructor, the virtual table would not have been created any function calls.
- Whereas destructor can be virtual.
- Let's first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
public:
~Base()
{
cout << "Base class destructor" << endl;
}
};

class Derived:public Base
{
public:
~Derived()
{
cout << "Derived class destructor" << endl;
}
};
```

```
int main()
{
Base* ptr = new Derived; //Base class pointer points to the
derived class object
delete ptr;
}
```

Output:

Base class Destructor

In the above example, delete ptr will only call the Base class destructor, which is undesirable because, then the object of the Derived class remains undeconstructed because its destructor is never called. Which leads to a **memory leak situation**.

To make sure that the derived class destructor is mandatory called, we make base class destructor as virtual

class Base

{

public:

virtual ~Base()

{

cout << "Base class destructor"<<endl;

}

};

Output:

Derived class Destructor

Base class Destructor

When we have a Virtual destructor inside the base class, then first Derived class's destructor is called and then the Base class's destructor is called, which is the desired behavior.