Unit –IV Pointers and Polymorphism in C++

Concepts of Pointer:

A pointer is a variable that holds a memory address of another variable where a value lives. A pointer is declared using the * operator before an identifier.

Pointer declaration

General syntax of pointer declaration is,

```
datatype *pointer_name;
```

Data type of a pointer must be same as the data type of the variable to which the pointer variable is pointing.

Here are a few examples:

```
int *ip    // pointer to integer variable

float *fp;    // pointer to float variable

double *dp;    // pointer to double variable

char *cp;     // pointer to char variable
```

Initialization of Pointer variable

Pointer Initialization is the process of assigning address of a variable to a pointer variable. Pointer variable can only contain address of a variable of the same data type.

Pointer operator

Operator	Operator Name	Purpose
*	Value at Operator	Gives Value stored at Particular address
&	Address Operator	Gives Address of Variable

In order to create pointer to a variable we use "*" operator and to find the address of variable we use "&" operator.

1. '&' operator is called as address Operator
2. **'*'** is called as **'Value at address'** Operator
3. '**Value at address'** Operator gives 'Value stored at Particular address.
4. **'Value at address'** is also called as **'Indirection Operator'**

For example,

```
#include<stdio.h>
```

```
int main()
{
int n = 20;
cout<< “\nThe address of n is "<<&n;
cout<< “\nThe Value of n is "<<n;
cout<< “\nThe Value of n is "<<*(&n);
}
```

Output:

```
The address of n is 1002
The Value of n is 20
The Value of n is 20
```

How *(&n) is same as printing the value of n ?

&n	Gives address of the memory location whose name is ‘n’
*	means value at Operator gives value at address specified by &n.
m = &n	Address of n is stored in m , but remember that m is not ordinary variable like ‘n’

So Ccompiler must provide space for it in memory.Below is Step by Step Calculation to compute the value –

```
m  =  * ( &n   )
   =  * ( Address of Variable 'n')
   =  * ( 1000 )
   =  Value at Address 1000
   =  20
```

Another example,

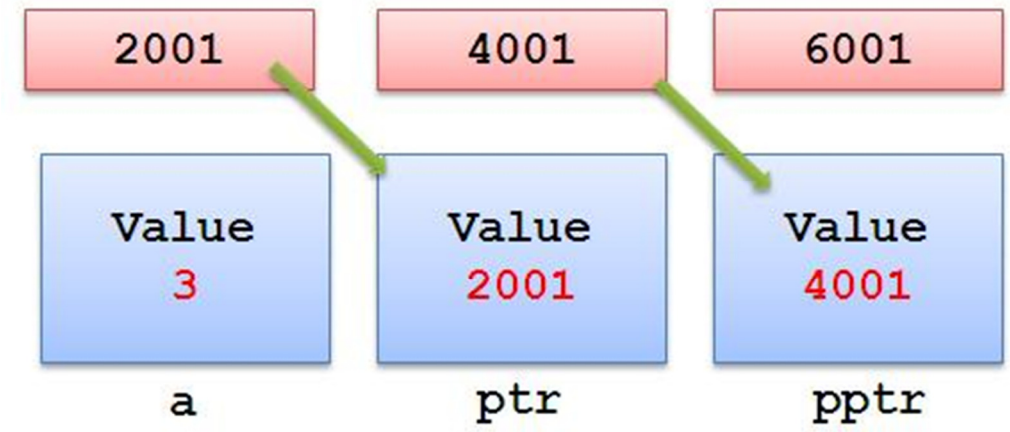
```
#include<stdio.h>

int main()
{
int a = 3;
int *ptr,**pptr;
```

```
ptr = &a;
pptr = &ptr;
return(0);
}
```

Explanation of Example

With reference to above program –



We have following associated points –

Point	Variable 'a'	Variable 'ptr'	Variable 'pptr'
Name of Variable	a	ptr	pptr
Type of Value that it holds	Integer	Address of 'a'	Address of 'ptr'
Value Stored	3	2001	4001
Address of Variable	2001	4001	6001

Pointer arithmetic.

Arithmetic Operations With Pointers

We can perform different arithmetic operations by using pointers. Increment, decrement, prefix, and postfix operations can be performed with pointers. The effects of these operations are shown in Table.

Table: Pointers and Arithmetic Operations

Data Type	Initial Address	Operation		Address After Operations		Required Bytes
int i = 2	4046	++	--	4048	4044	2
char c = 'x'	4053	++	--	4054	4052	1
float f =2.2	4058	++	--	4062	4054	4
long l = 2	4060	++	--	4064	4056	4

From the above table, while referring to the first entry, we can observe that on increment of the pointer variable for integers, the address is incremented by two; that is, 4046 is the original address and on increment, its value will be 4048, because integers require two bytes. Similarly, when the pointer variable for integer is decreased, its address 4048 becomes 4046.

Similarly, for characters, floating point numbers and long integers require 1, 4, and 4 bytes, respectively. After the effect of increment and decrement, the memory locations are shown in Table.

The following program explains the increase and decrease of pointers:

The following program explains the increase and decrease of pointers:

13.4 Program on pointer incrementation and decrementation.

```
#include<iostream.h>
#include<conio.h>
int main()
{
   clrscr();
   int x=10;
   int *p;
   p=&x;
   cout<<"\n Address of p:"<<unsigned(p);
   p=p+4;
   cout<<"\n Address of p:"<<unsigned(p);
   p=p-2;
   cout<<"\n Address of p:"<<unsigned(p);
   return 0;
}
```

OUTPUT

```
Address of p:65524
Address of p:65532
Address of p:65528
```

Explanation: In the above program, p holds the address of x. The initial address of x is 65524. When p is incremented by 4, it means that the address is increased by 8, because each integer needs two bytes. Here, the address obtained is 65532. Similarly, when the address of x is decreased to 2, the address finally obtained is 65528.

13.5 Program on changing the values of variables using pointer.

```
#include<iostream.h>
#include<conio.h>
int main()
{
   clrscr();
   int x=10;
   int *p;
   p=&x;
   cout<<"\n Value of x:"<<*p;
   *p=*p+10;
   cout<<"\n Value of x:"<<*p;
   *p=*p-2;
   cout<<"\n Value of x:"<<*p;
   return 0;
}
```

OUTPUT

```
Value of x:10
Value of x:20
Value of x:18
```

Explanation: In the above program using pointers, the value of the variable x is first increased and then decreased.

Pointer To Object

Similar to variables, objects also have an address. A pointer can point to a specified object. The following program illustrates this:

13.11 Write a program to declare an object and pointer to the class. Invoke the member functions using pointer.

```
#include<iostream.h>
#include<conio.h>
class Bill
{
   int qty;
   float price;
   float amount;
   public :
   void getdata (int a, float b, float c)
   {
      qty=a;
      price=b;
      amount=c;
   }
   void show()
   {
   cout<<"Quantity : " <<qty <<"\n";
   cout<<"Price : " <<price <<"\n";
   cout<<"Amount : " <<amount <<"\n";
   }
};
int main()
{
   clrscr();
   Bill s;
   Bill *ptr =&s;
   ptr->getdata(45,10.25,45*10.25);
   (*ptr).show();
   return 0;
}
```

OUTPUT

```
Quantity : 45
Price : 10.25
Amount : 461.25
```

Explanation: In the above program, the class Bill contains two float and one int members. The class Bill also contains the member function getdata() and show() to read and display the data. In function main(), s is an object of class Bill, and ptr is a pointer of the same class. The address of object s is assigned to pointer ptr. Using pointer ptr with arrow operator (->) and dot operator (.), members and functions are invoked. The statements used for invoking functions are as given below.

```
ptr->getdata (45,10.25,45*10.25);
(*ptr).show();
```

Here, both the pointer declarations are valid. In the second statement, ptr is enclosed in brackets, because the dot operator (.) has higher precedence as compared with the indirection operator (*). The output of the program is as shown above.

13.12 Write a program to create dynamically an array of objects of class type. Use new operator.

```
#include<iostream.h>
#include<conio.h>
class Bill
{
   int qty;
   float price;
   float amount;
   public :
   void getdata (int a, float b, float c)
   {
      qty=a;
      price=b;
      amount=c;
   }
   void show()
   {
   cout<<"Quantity : " <<qty <<"\n";
   cout<<"Price : " <<price <<"\n";
   cout<<"Amount : " <<amount <<"\n";
   }
};
int main()
{
   clrscr();
   Bill *s= new Bill[2];
   Bill *d =s;
   int x,i;
   float y;
   for (i=0;i<2;i++)
   {
   cout<<"\nEnter Quantity and Price : ";
   cin>>x >>y;
   s->getdata(x,y,x*y);
   s++;
```

```
    }
    for (i=0;i<2;i++)
    {
    cout<<endl;
    d->show();
    d++;
    }
    return 0;
}
```

OUTPUT

```
Enter Quantity and Price : 5 5.3
Enter Quantity and Price : 8 9.5
Quantity : 5
Price : 5.3
Amount : 26.5
Quantity : 8
Price : 9.5
Amount : 76
```

Explanation: In the above program, the class Bill is similar to that in the previous example. In main(), using new memory allocation operator, the memory required for two objects is allocated to pointer s, that is, 10 bytes. The first for loop accepts the data through the keyboard. Immediately after this, the data are sent to the member function getdata(). The pointer s is incremented. After incrimination, it points to the next memory location of its type. Thus, two records are read through the keyboard. The second for loop is used to display the contents on the screen. Here, the function show() is invoked. The logic used is similar to that used in the first loop. The functions are invoked using pointers, and this has been explained in the previous example.

‘this’ pointer

- C++ provides a keyword 'this', which represents the current object and passed as a hidden argument to all member functions.
- The **this** pointer is a constant pointer that holds the memory address of the current object.
- 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.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

C++ program using this pointer

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

```
#include<conio.h>
class sample
{
   int a,b;
   public:
      void input(int a,int b)
      {
         this->a=a+b;
         this->b=a-b;
      }
      void output()
      {
         cout<<"a = "<<a<<endl<<"b = "<<b;
      }
};

int main()
{
   sample x;
   x.input(5,8);
   x.output();
   getch();
   return 0;
}
```

Output:
A=13
B= -3

Write a program to declare a class test having data members name and marks of student. Accept and display data using ‘this’ pointer.

```
#include<iostream.h>
#include<conio.h>
class test
{
   Char name[30];
   int mark;
   public:
void get()
{
Cout<<”\n Enter name and marks”;
Cin>>name>>mark;
}
Void display()
{
Cout<<”\n Name=”<<this - >name;
Cout<<”\n Marks=”<<this - >mark;
}
};
```

```
Void main()
{
Test t1;
T1.get();
T1.display();
Getch();
}
```

Output:
Enter name and marks
Pragati 83

Name= Pragati
Marks=83

13.14 Write a program to enter name and age of two persons. Find the elder person. Use this pointer.

```
#include<iostream.h>
#include<conio.h>
   class name
   {
      char str[15];
      int age;
      public:
   void input()
   {
      cout<<"\n Enter Name and age : ";
      cin>>str;
      cin>>age;
   }
   void show()
   {
      cout<<"\n\nElder person ";
      cout<<"\n Name: "<<str;
      cout<<"\n Age: "<<age;
   }
   name display( name x)
   {
      cout<<"\n\n Contents of object n1 (this pointer)";
      cout<<"\n Name:" <<this->str;
      cout<<"\n Age:" <<this->age;
      cout<<"\n\n Contents of object n2 (x)";
      cout<<"\n Name: " <<x.str;
      cout<<"\n Age: " <<x.age;
```

```
        if (this->age>x.age)
        return *this;
        else
        return x;
    }
};
void main()
{
clrscr();
name n,n1,n2;
n1.input();
n2.input();
n=n1.display(n2);
n.show();
}
```

OUTPUT

```
Enter Name and age : Mahesh 25
Enter Name and age : Suresh 30
Contents of object n1(this pointer)
Name : Mahesh
Age : 25
Contents of object n2 (x)
Name : Suresh
Age : 30
Elder person
Name : Suresh
Age : 30
```

Explanation: The above program is similar to the previous one. Here, contents of explicitly and implicitly passed objects are displayed. The this pointer points to the hidden argument (implicit argument). The if statement compares the member variable age of both the objects and returns the greater one. The object n collects the return value of function display() and calls the function show(). The function show() displays the contents of object n. Working of this pointer is illustrated in Figure.

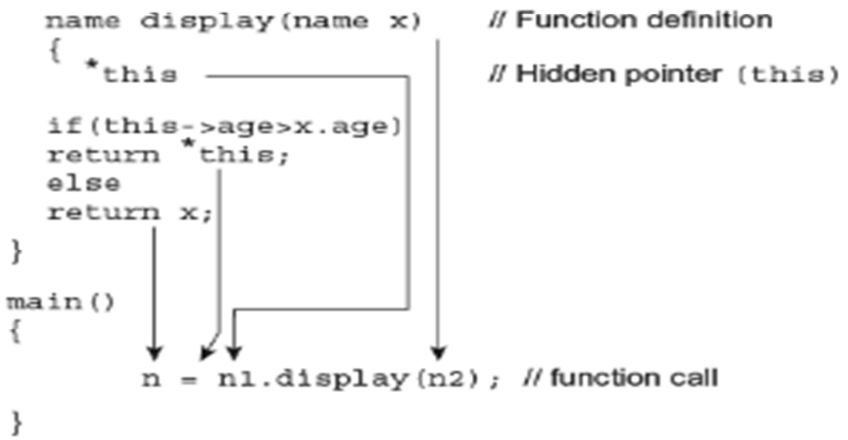


Fig: Working of this pointer

Pointer to derived class

- Pointers can be declared to the point base or derived class.
- A base class pointer can point to object of both the base and derived class.
- Derived class pointer can point to only derived class.

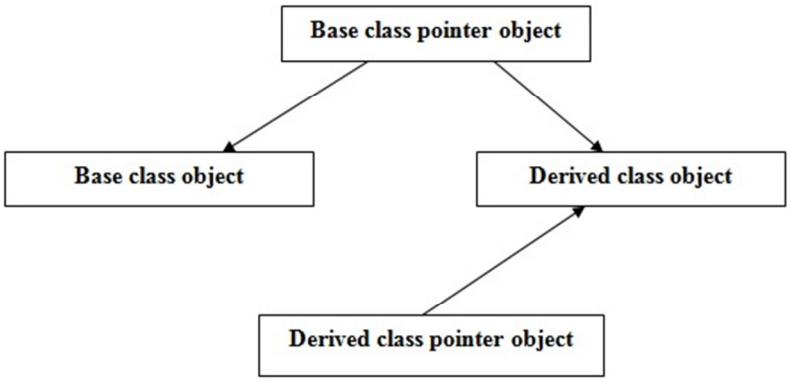


Fig: Type Compatibility of base and derived class

Program 4.6: Pointer to derived classes.

```
#include<iostream.h>
class base
{
    public:
    int n1;
    void show()
    {
       cout<<"\nn1 = "<<n1;
    }
};
class derive : public base
{
    public:
    int n2;
    void show()
    {
       cout<<"\nn1 = "<<n1;
       cout<<"\nn2 = "<<n2;
    }
};
int main()
{
   base b;
   base *bptr;      //base pointer
   cout<<"Pointer of base class points to it";
   bptr=&b;         //address of base class
   bptr->n1=44;             //access base class via base pointer
   bptr->show();
   derive d;
   cout<<"\n";
   bptr=&d;         //address of derive class
```

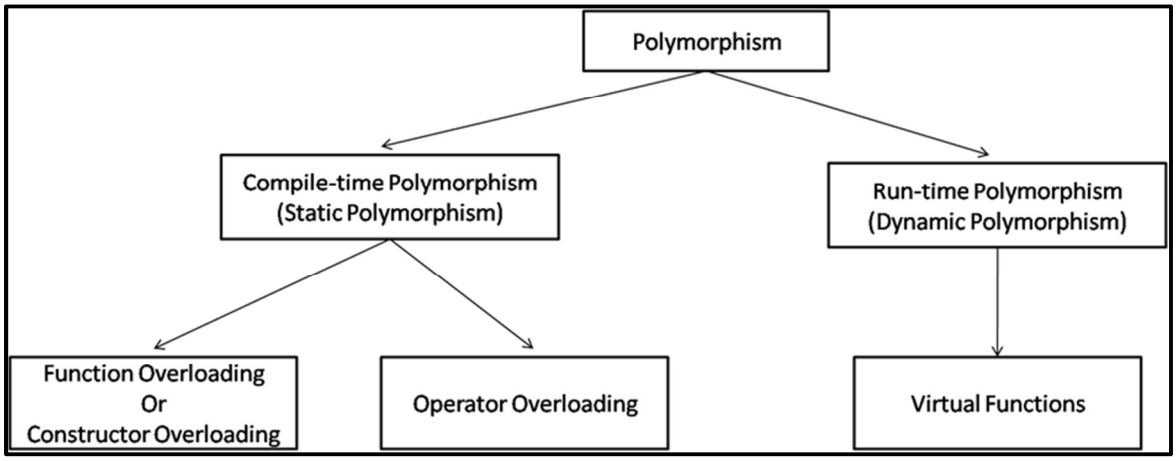
```
    bptr->n1=66;            //access derive class via base pointer
    bptr->show();
    return 0;
}
```

Output:
Pointer of base class points to it
n1 = 44
Pointer of base class points to derive class
n1=66

Introduction of Polymorphism

- Polymorphism it is the ability to form more than one form.
- Polymorphism is the ability of a message to be processed in more than one way. A different member function can be invoked at different instances depending on the object which is invoking the function.

Polymorphism is of two types:
1) Compile-time polymorphism
2) Run-time polymorphism



Compile-time polymorphism / Early Binding / Static Binding

- In this polymorphism objects are created at the time of compilation.
- Its further divide into two parts function overloading and operator overloading.

Function overloading

Two or more functions having same name but different argument(s) are known as overloaded functions.

Program 4.7: Program for function overloading

```
#include <iostream>
using namespace std;
long add(long, long);
float add(float, float);
 int main()
{
   long a, b, x;
   float c, d, y;

   cout << "Enter two integers\n";
   cin >> a >> b;

   x = add(a, b);

   cout << "Sum of integers: " << x << endl;
    cout << "Enter two floating point numbers\n";
   cin >> c >> d;
    y = add(c, d);
    cout << "Sum of floats: " << y << endl;
    return 0;
}
 long add(long x, long y)
{
   long sum;
    sum = x + y;
    return sum;
}

float add(float x, float y)
{
   float sum;
    sum = x + y;
```

```
    return sum;
}
Enter two integers
10      20
Sum of integers: 30
Enter two floating point numbers
1.1         2.2
Sum of floats: 3.3
```

Program 4.7: Write a program to print area of circle, rectangle and triangle using function overloading

```
#include<iostream.h>
#include<stdlib.h>
#include<conio.h>
#define pi 3.14

class fn {
public:
    void area(int); //circle
    void area(int, int); //rectangle
    void area(float, int, int); //triangle
};

void fn::area(int a) {



    cout << "Area of Circle:" << pi * a*a;
}

void fn::area(int a, int b) {
    cout << "Area of rectangle:" << a*b;
```

```
}

void fn::area(float t, int a, int b) {
   cout << "Area of triangle:" << t * a*b;
}

void main() {
   int ch;
   int a, b, r;
   clrscr();
   fn obj;
   cout << "\n\t\tFunction Overloading";
   cout << "\n1.Area of Circle\n2.Area of Rectangle\n3.Area of Triangle\n4.Exit\n:?;
         cout << ?Enter your Choice : ";
         cin>>ch;

   switch (ch) {
      case 1:
         cout << "Enter Radious of the Circle:";
         cin>>r;
         obj.area(r);
         break;
      case 2:
         cout << "Enter Sides of the Rectangle:";
         cin >> a>>b;
         obj.area(a, b);
         break;
      case 3:
         cout << "Enter Sides of the Triangle:";
         cin >> a>>b;
         obj.area(0.5, a, b);
         break;
      case 4:
```

```
        exit(0);
    }
    getch();
}
```

Output:

```
Function Overloading
1. Area of Circle
2. Area of Rectangle
3. Area of Triangle
4. Exit
Enter Your Choice: 2

Enter the Sides of the Rectangle: 5 5

Area of Rectangle is: 25

1. Area of Circle
2. Area of Rectangle
3. Area of Triangle
4. Exit
Enter Your Choice: 4
```

Operator overloading

Almost all arithmetic operator can be overloaded to perform arithmetic operation on user-defined data type.

Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.

An operator can be overloaded by defining a function to it. The function for operator is declared by using the operator keyword followed by the operator.

There are two types of operator overloading in C++

- Binary Operator Overloading
- Unary Operator Overloading

Program 4.8: C++ program for unary minus (-) operator overloading.

```
#include<iostream.h>
class NUM
{
   private:
      int n;

   public:
      //function to get number
      void getNum(int x)
      {
         n=x;
      }
      //function to display number
      void dispNum(void)
      {
         cout << "value of n is: " << n;
      }
      //unary - operator overloading
      void operator - (void)
      {
         n=-n;
      }
};

int main()
{
   NUM num;
   num.getNum(10);
   -num;
   num.dispNum();
   cout << endl;
   return 0;
```

```
}
```

Output:

```
value of n is: -10
```

Program 4.9: C++ program for unary increment (++) and decrement (--) operator overloading.

```
#include<iostream.h>
class NUM
{
   private:
      int n;

   public:
      //function to get number
      void getNum(int x)
      {
         n=x;
      }
      //function to display number
      void dispNum(void)
      {
         cout << "value of n is: " << n;
      }
      //unary ++ operator overloading
      void operator ++ (void)
      {
         n=++n;
      }
      //unary -- operator overloading
      void operator -- (void)
      {
         n=--n;
      }
```

```
};
int main()
{
   NUM num;
   num.getNum(10);

   ++num;
   cout << "After increment - ";
   num.dispNum();
   cout << endl;

   --num;
   cout << "After decrement - ";
   num.dispNum();
   cout << endl;
   return 0;
}
```

Output:

After increment - value of n is: 11

After decrement - value of n is: 10

Program 4.10: C++ Program to Add Two Objects using Operator Overloading

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

class sum {
   public:
   int x, y, z;
   void getdata(int a, int b)
   {
      x=a;
      y=b;
```

```
    }

    void display()
    {
        cout<<"\nSum of X:"<<x;
        cout<<"\nSum of Y:"<<y;
    }
    void operator+(sum &);
};

void sum::operator+(sum &ob) {
    x=x+ob.x;
    y=y+ob.y;
    display();
}

void main()
{
    sum ob1, ob2;
    clrscr();
    ob1.getdata(10,20);
    ob2.getdata(20,30);
    ob1+ob2;
    getch();
}
Output
Sum of X:30
Sum of Y:50
```

Overloading of binary operators

Program 4.11: Binary Operator Overloading Example Program to add two complex numbers

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

```
class complex {
   int a, b;
public:

   void getvalue() {
      cout << "Enter the value of Complex Numbers a,b:";
      cin >> a>>b;
   }

   complex operator+(complex ob) {
      complex t;
      t.a = a + ob.a;
      t.b = b + ob.b;
      return (t);
   }

   complex operator-(complex ob) {
      complex t;
      t.a = a - ob.a;
      t.b = b - ob.b;
      return (t);
   }

   void display() {
      cout << a << "+" << b << "i" << "\n";
   }
};

void main() {
   clrscr();
   complex obj1, obj2, result, result1;
   obj1.getvalue();
   obj2.getvalue();
   result = obj1 + obj2;
```

```
    result1 = obj1 - obj2;
    cout << "Input Values:\n";
    obj1.display();
    obj2.display();
    cout << "Result:";
    result.display();
    result1.display();

    getch();
}
```

Output:

```
Enter the value of Complex Numbers a, b
4               5
Enter the value of Complex Numbers a, b
2               2
Input Values
4 + 5i
2 + 2i
Result
6 +  7i
2 +  3i
```

Function Overriding

When a base class and sub class contains a function with the same signature, and the function is called with base class object, then the function in derived class executes and the function in base class is said to be overridden. This is known as function overriding. Following program demonstrates function overriding:

Program 4.11: program demonstrates function overriding:

```
#include<iostream>
class A
{
protected:
int x;
public:
void show()
```

```
{
cout<<“x = “<<x<<endl;
}
};
class B : public A
{
protected:
int y;
public:
B(int x, int y)
{
this->x = x;
this->y = y;
}
void show()
{
cout<<“x = “<<x<<endl;
cout<<“y = “<<y<<endl;
}
};
int main()
{
A objA;
B objB(30, 20);
objB.show();
return 0;
}
```

Output for the above program is:

x = 30

y = 20

Function Overloading VS Function Overriding

	Function Overloading	Function Overriding
Inheritance	Overloading can occur without inheritance.	Overriding of functions occurs when one class is inherited from another

		class.
Function Signature	Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ.	In overriding, function signatures must be same.
Scope of functions	Overloaded functions are in same scope.	Overridden functions are in different scopes
Behaviour of functions	Overloading is used to have same name functions which behave differently depending upon parameters passed to them.	Overriding is needed when derived class function has to do some added or different job than the base class function.

Rules for operator overloading.

Every programmer knows the concept of operation overloading in C++. Although it looks simple to redefine the operators in operator overloading, there are certain restrictions and limitation in overloading the operators. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be overloaded.

2. The overloaded operator must have at least one operand that is of user defined type.

3. We cannot change the basic meaning of an operator. That is to say, We cannot redefine the plus(+) operator to subtract one value from the other.

1. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
2. There are some operators that cannot be overloaded like
    a. size of operator(sizeof)
    b. membership operator(.)
    c. pointer to member operator(.*)
    d. scope resolution operator(::)
    e. conditional operators(?:)

6. We cannot use "friend" functions to overload certain operators. However, member function can be used to overload them. Friend Functions cannot be used with
    a. assignment operator(=)
    b. function call operator(())
    c. subscripting operator([])
    d. class member access operator(->)

7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevent class).

8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

10. Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.

Run time polymorphism/ Late Binding/ Dynamic 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.
- Runtime Polymorphism is a form of polymorphism at which function binding occurs at runtime.
- You can have a parameter in subclass, same as the parameters in its super classes with the same name. Virtual keyword is used in superclass to call the subclass. The function call takes place on the run time, not on the compile time.

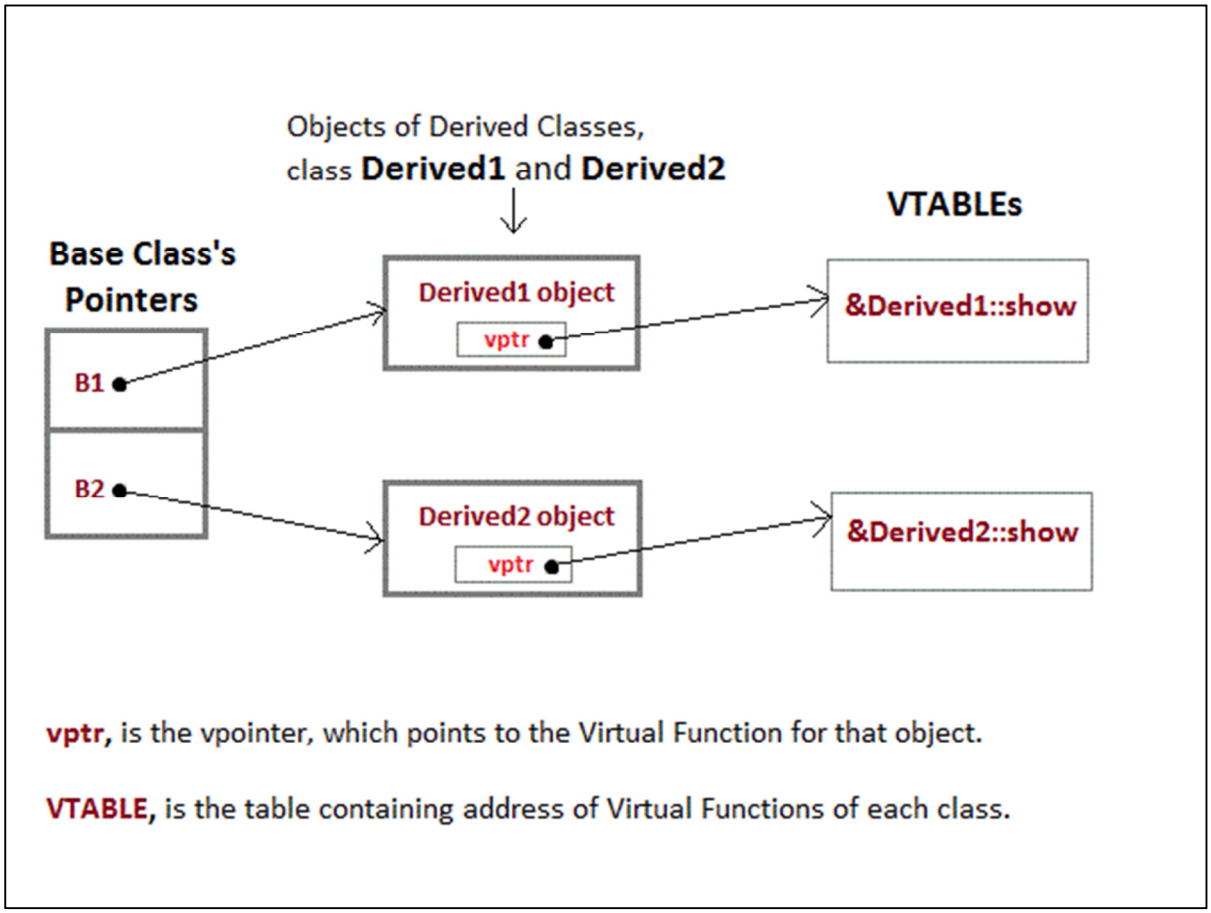


Figure: Mechanism of late binding

Virtual functions

- Virtual Function is a function in base class, which is overrided 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.

```
class Base
{
```

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

int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;
 b->show();     //Late Binding Ocuurs
}
Output:
Derived Class
```

Difference between Runtime Polymorphism and Compile time Polymorphism

Sr. No.	Compile time Polymorphism	Run time Polymorphism
1	In Compile time Polymorphism, call is resolved by the **compiler**.	In Run time Polymorphism, call is **not** resolved by the compiler.
2	It is also known as **Static binding, Early binding** and **overloading** as well.	It is also known as **Dynamic binding, Late binding** and **overriding** as well.
3	**Overloading** is compile time polymorphism where more than one methods share the same name with different parameters or signature and different return type.	**Overriding** is run time polymorphism having same method with same parameters or signature, but associated in a class & its subclass.

4	It is achieved by **function** overloading and **operator** overloading.	It is achieved by **virtual functions** and **pointers**.
5	It provides **fast execution** because known early at compile time.	It provides **slow execution** as compare to early binding because it is known at runtime.
6	Compile time polymorphism is **less flexible** as all things execute at compile time.	Run time polymorphism is **more flexible** as all things execute at run time.

Rules for virtual functions

- Virtual functions must be members of a class.
- Virtual functions must be created in public section so that objects can access them.
- When virtual function is defined outside the class, virtual keyword is required only in the function declaration. Not necessary in the function definition.
- Virtual functions cannot be static members.
- Virtual functions must be accessed using a pointer to the object.
- A virtual function cannot be declared as a friend of another class.
- Virtual functions must be defined in the base class even though it does not have any significance.
- The signature of virtual function in base class and derived class must be same.
- A class must have a virtual destructor but it cannot have a virtual constructor.

Pure virtual function

Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function,

```
virtual void f() = 0;
```

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

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

```
};


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

Output :

Implementation of Virtual Function in Derived class

Question Bank:

2 Marks

1. State two pointer operators.
2. What is a pointer? Write down the general syntax of it's declaration.
3. Enlist any four operators which can not be overloaded.
4. List types of polymorphism

4 Marks

1. Explain the concept of pointer to object with suitable example.
2. Distinguish between run-time polymorphism & compile-time polymorphism.
3. State any four rules for operator overloading.
4. Write a program using function overloading to swap 2 integer numbers & swap 2 float numbers.
5. Explain the concept of 'this' pointer.
6. Explain virtual function with suitable example.
7. Write a program to declare a class distance having data members feet & inches. Overload unary '_' operator so that when it is used with object of this class, it will decrement values of inches by 1.