# 4.1 Concepts of Pointer

## Pointers in C++

Pointer is a variable in C++ that holds the address of another variable. They have data type just like variables, for example an integer type pointer can hold the address of an integer variable and an character type pointer can hold the address of char variable.

## Syntax of pointer

```
data_type *pointer_name;
```

### How to declare a pointer?

```
/* This pointer p can hold the address of an integer
 * variable, here p is a pointer and var is just a
 * simple integer variable
 */
int *p, var
```

# Pointer operator * in C++

C++ provides two pointer operators, which are Address of Operator (&) and Indirection Operator (*). A pointer is a variable that contains the address of another variable or you can say that a variable that contains the address of another variable is said to "point to" the other variable. A variable can be any data type including an object, structure or again pointer itself.

The indirection Operator (*), and it is the complement of &. It is a unary operator that returns the value of the variable located at the address specified by its operand. For example,

## Example

```
#include <iostream>

using namespace std;

int main () {

   int  var;

   int  *ptr;

   int  val;

   var = 3000;


   // take the address of var

   ptr = &var;


   // take the value available at ptr
```

```
   val = *ptr;

   cout << "Value of var :" << var << endl;

   cout << "Value of ptr :" << ptr << endl;

   cout << "Value of val :" << val << endl;

   return 0;

}
```

## Output

When the above code is compiled and executed, it produces the following result −

```
Value of var : 3000
Value of ptr : 0xbff64494
Value of val : 3000
```

# Pointer Arithmetic

As you understood pointer is an address which is a numeric value; therefore, you can perform arithmetic operations on a pointer just as you can a numeric value. There are four arithmetic operators that can be used on pointers: ++, --, +, and -

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

```
ptr++
```

the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

## Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

Live Demo

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have array address in pointer.
```

```
   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;

      // point to the next location
      ptr++;
   }

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

## Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

Live Demo

```
#include <iostream>

using namespace std;
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have address of the last element in pointer.
   ptr = &var[MAX-1];

   for (int i = MAX; i > 0; i--) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the previous location
      ptr--;
   }
```

```
   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Address of var[3] = 0xbfdb70f8
Value of var[3] = 200
Address of var[2] = 0xbfdb70f4
Value of var[2] = 100
Address of var[1] = 0xbfdb70f0
Value of var[1] = 10
```

# Pointers to Objects

A variable that holds an address value is called a pointer variable or simply pointer.
Pointer can point to objects as well as to simple data types and arrays.
sometimes we dont know, at the time that we write the program , how many objects we want to creat. when this is the case we can use new to creat objects while the program is running. new returns a pointer to an unnamed objects. lets see the example of student that wiil clear your idea about this topic

```
#include <iostream>
#include <string>
using namespace std;
class student
{
private:
        int rollno;
        string name;
public:
        student():rollno(0),name("")
        {}
        student(int r, string n): rollno(r),name (n)
        {}
        void get()
        {
                cout<<"enter roll no";
                cin>>rollno;
                cout<<"enter  name";
                cin>>name;
        }
        void print()
        {
                cout<<"roll no is "<<rollno;
                cout<<"name is "<<name;
        }
};
void main ()
{
        student *ps=new student;
        (*ps).get();
        (*ps).print();
        delete ps;
}
```

# ‘this’ pointer in C++

09-08-2012

To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.
Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?
The compiler supplies an implicit pointer along with the names of the functions as ‘this’.
The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).
For a class X, the type of this pointer is ‘X* ‘. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *’ (see this GFact)
In the early version of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.
C++ lets object destroy themselves by calling the following code :

filter_none

brightness_4

```
delete this;
```

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.
Following are the situations where ‘this’ pointer is used:

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

filter_none

edit

play_arrow

brightness_4

```
#include<iostream>
using namespace std;

/* local variable is same as a member's name */
class Test
{
private:
   int x;
public:
   void setX (int x)
   {
```

```
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable '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
```

For constructors, initializer list can also be used when parameter name is same as member's name.

**2) To return reference to the calling object**

filter_none
edit
play_arrow
brightness_4

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
    // Some processing
    return *this;
}
```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

filter_none
edit
play_arrow
brightness_4

```
#include<iostream>
using namespace std;

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);

  // Chained function calls.  All calls modify the same object
  // as the same object is returned by reference
  obj1.setX(10).setY(20);

  obj1.print();
  return 0;
}
```

Output:

```
x = 10 y = 20
```

**Exercise:**

Predict the output of following programs. If there are compilation errors, then fix them.

**Question 1**

filter_none
edit
play_arrow
brightness_4

```
#include<iostream>
using namespace std;

class Test
{
private:
  int x;
public:
  Test(int x = 0) { this->x = x; }
  void change(Test *t) { this = t; }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj(5);
  Test *ptr = new Test (10);
  obj.change(ptr);
  obj.print();
  return 0;
}
```

**Question 2**

filter_none
edit
play_arrow

brightness_4

```
#include<iostream>
using namespace std;

class Test
{
private:
  int x;
  int y;
public:
  Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
  static void fun1() { cout << "Inside fun1()"; }
  static void fun2() { cout << "Inside fun2()"; this->fun1(); }
};

int main()
{
  Test obj;
  obj.fun2();
  return 0;
}
```

**Question 3**

filter_none

edit

play_arrow

brightness_4

```
#include<iostream>
using namespace std;

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;
  obj1.setX(10).setY(20);
  obj1.print();
  return 0;
}
```

**Question 4**

filter_none
edit
play_arrow
brightness_4

```
#include<iostream>
using namespace std;

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

int main()
{
  Test obj;
  obj.destroy();
  obj.print();
  return 0;
}
```

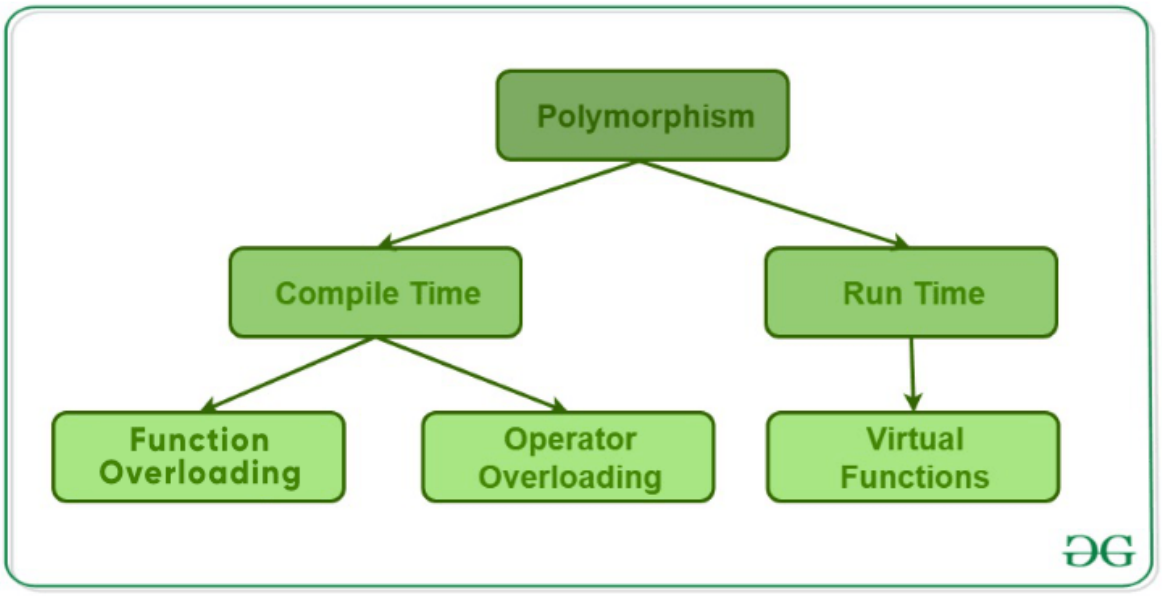
# Polymorphism in C++

22-05-2017

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

**In C++ polymorphism is mainly divided into two types:**

- Compile time Polymorphism
- Runtime Polymorphism




1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

- **Function Overloading**: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.
  Rules of Function Overloading

```
// C++ program for function overloading
#include <bits/stdc++.h>

using namespace std;
class Geeks
{
    public:

    // function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }

    // function with same name but 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }

    // function with same name and 2 int parameters
    void func(int x, int y)
    {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};
```

```
int main() {

    Geeks obj1;

    // Which function is called will depend on the parameters passed
    // The first 'func' is called
    obj1.func(7);

    // The second 'func' is called
    obj1.func(9.132);

    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
```

**Output:**

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.

# Functions that cannot be overloaded in C++

In C++, following function declarations **cannot** be overloaded.
1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
#include<iostream>
int foo() {
  return 10;
}

char foo() {
  return 'a';
}

int main()
{
   char x = foo();
   getchar();
   return 0;
}
```

2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration. For example, following program fails in compilation.

```
#include<iostream>
class Test {
   static void fun(int i) {}
   void fun(int i) {}
};

int main()
{
   Test t;
   getchar();
   return 0;
}
```

3) Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)()); // redeclaration of h(int())
```

5) Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error *"redefinition of `int f(int)' "*

**Example:**

```
#include<iostream>
#include<stdio.h>
using namespace std;
int f ( int x) {
    return x+10;
}
int f ( const int x) {
    return x+10;
}
int main() {
  getchar();
  return 0;
}
```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within

a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, "pointer to T," "pointer to const T," and "pointer to volatile T" are considered distinct parameter types, as are "reference to T," "reference to const T," and "reference to volatile T."
6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *"redefinition of `int f(int, int)' "*

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

using namespace std;

int f ( int x, int y) {
    return x+10;
}

int f ( int x, int y = 10) {
    return x+y;
}

int main() {
  getchar();
  return 0;
}
```

**Operator Overloading**: C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.
**Example**:

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag
= i;}

    // This is automatically called when '+' is used
with
```

```
    // between two Complex objects
    Complex operator + (Complex const &obj) {
         Complex res;
         res.real = real + obj.real;
         res.imag = imag + obj.imag;
         return res;
    }
    void print() { cout << real << " + i" << imag <<
endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to
"operator+"
    c3.print();
}
```

**Output:**

```
12 + i9
```

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit this link.

2. **Runtime polymorphism**: This type of polymorphism is achieved by Function Overriding.
   - **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding

#include <bits/stdc++.h>
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 () //print () is already virtual function in derive
                  //we could also declared as virtual void print (
    { cout<< "print derived class" <<endl; }

    void show ()
    { cout<< "show derived class" <<endl; }
};

//main function
int main()
{
    base *bptr;
    derived d;
    bptr = &d;

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

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

    return 0;
}
```

**Output:**

```
print derived class
show base class
```

## C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- 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. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

# Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

**Rules of Virtual Function**

- ✓ Virtual functions must be members of some class.
- ✓ Virtual functions cannot be static members.
- ✓ They are accessed through object pointers.
- ✓ They 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 the overloaded functions.
- ✓ We cannot have a virtual constructor, but we can have a virtual destructor
- ✓ Consider the situation when we don't use the virtual keyword.

```
#include <iostream>
using namespace std;
class A
{
   int x=5;
    public:
    void display()
    {
       std::cout << "Value of x is : " << x<<std::endl;
    }
};
class B: public A
{
    int y = 10;
    public:
    void display()
    {
       std::cout << "Value of y is : " <<y<< std::endl;
    }
};
int main()
{
    A *a;
    B b;
    a = &b;
   a->display();
    return 0;
}
```

**Output:**

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

## Virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

**Output:**

```
Derived Class is invoked
```

# Pure Virtual Function

- ✓ A virtual function is not used for performing any task. It only serves as a placeholder.
- ✓ When the function has no definition, such function is known as "**do-nothing**" function.
- ✓ The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- ✓ A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.

- ✓ The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

**Pure virtual function can be defined as:**

1. `virtual void display() = 0;`

**Let's see a simple example:**

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." <
< std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
   bptr->show();
   return 0;
}
```

**Output:**

```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.