What is pointer and how to declare it? Write the features of pointers?

A pointer is a memory variable that stores the address of another variable. Pointer can have any name that is legal for other variables, and it is declared in the same way as other variables, but it is always denoted by *‘*’* operator.

Pointer declaration

Pointer varible can be declared as follows:

type *variable_name;

where type is the data type pointed to by the pointer and variable name is pointer variable name.
For example:

*int *x;*
*float *f;*
*char *c;*

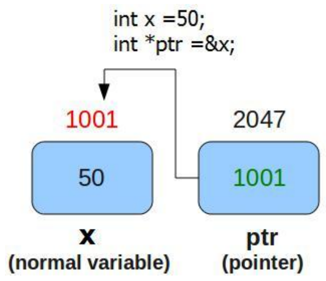
1. In the first statement , „x‟ is an integer pointer, and informs the compiler that it holds the address address of any integer variable.
2. Normal variables provide direct access to their own variables; where as pointer indirectly access the value of a variable to which it points.
3. The indirection (*) operator is used in two distinct ways with pointers: declaration and dereference.
4. When a pointer is declared , the star indicatesthat it is pointer, not a normal variable.
5. When a pointer is dereferenced, the indirection operator indicates that the value at that memory location stored in the pointer.
6. The „&‟ operator is the address of operator and it represents the address of a variable. The address of a variable is always a whole number.

Example

```
#include<iostream.h>
int main()
{
        int *p;
        int x=50;
        p= &x;
        cout<<"value of x="<<x<<"\taddress of x="<<&(unsigned)x;       //using cout()
       cout<<"value of x="<<*p<<"\taddress of x="<<(unsigned)p;        //using pointers
        return 0;
}
```

Output:

```
value of x=50          address of x=1001
value of x=50          address of x=1001
```



Features of pointers

1. Pointers save memory space.
2. Execution time with pointers is faster, because data is manipulated with address.
3. Memory is accessed efficiently with pointers. The pointers assigns as well as releases the memory space.
4. Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
5. Pointers are used for file handling.
6. Pointers are used to allocate memory in a dynamic manner.
7. In C++, a pointer declared to a derived class cannot access the object of a derived class.

How to create pointer to class?

(OR)

Explain about class pointers?

We know that pointer is a variable that holds the address of another variable. In the same way, we can also define *pointer to class*. Here, the starting address of the member variables can be accessed. Thus such variables are called class pointers.

Example:

```
#include<iostream.h>
#include<conio.h>
class Man
{
   public:
     char name[25];
     int age;
};

int main()
{
        Man m = {“Vijayanand”,30};
        Man *ptr;
        ptr= &m;
        clrscr();
        cout<< "\nName= "<<m->name;
        cout<< "\tAge= "<<m->age;
        return 0;
}
```

Output:

```
Name=Vijayanand     Age=30
```

How to create pointer to object? Explain with an example?

(OR)

Explain object pointers? Explain with an example?

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

Example:

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

```
#include<conio.h>
class Bill
{
        int qty;
        float price, amount;
    public:
           void getdata (int a, float b)
           {
                    qty = a;
                    price = b;
                    amount = a*b;
           }
           void show()
           {
                    cout<< "\nQuantity:"<<qty;
                    cout<< "\nPrice: "<<price;
                    cout<< "\nAmount: "<<amount;
           }
};

int main()
{
           clrscr();
           Bill b;
           Bill *bptr = &b;
           bptr->getdata(5,20);
           (*bptr).show();
           return 0;
}
```

Output:

```
Quantity: 5
Price: 20
Amount: 100"
```

Explain about *this* pointer with an example?

Every object in C++ has access to its own address through an unseen 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.

Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a ***this*** pointer. Let us understand *this* pointer with following example:

```
#include<iostream.h>
#include<conio.h>
class Number
{
        int x;
    public:
           void input( )
           {
                   cout<< "\nEnter a number:";
                   cin>>x;
           }
           void show()
           {
                   cout<< "\nThe Minimum number is :"<<x;
           }

           Number min( Number n2 )
           {
                   if(n2.x<x)
                           return n2;
                   else
                           return *this;
           }
};

int main()
{
           clrscr();
           Number n,n1,n2;
           n1.input();
           n2.input();
           n= n1.min(n2);
           n.show();
           return 0;
}
```

Output

```
Enter a number: 3
Enter a number: 5
The minimum number is 3.
```

Pointer to Derived classes and Base class

It is also possible to declare a pointer that points to the base class as well as the derived class. One pointer can point to different classes. For example, X is a base class and Y is a derived class. The pointer pointing to X can also point to Y.

Example:

```
#include<iostream.h>
#include<conio.h>
class A
{
  public:
      int b;
      void show( )                          //base class member function
     {
          cout <<"\nb="<<b;
     }
};

class B: public A
{
     public:
          int d;
        void show( )                //derived class member function
        {
             cout <<"\nd="<<d;
         }
};

int main()
{
    clrscr():
    A *bp; A
    base; bp=
    &base;
//    bp->b=100;                   //Not accessible
    cout<<"bp points to the base class object\n";
    bp->show();
    B b;
    cout<<"\nbp points to the derived class object\n";
    bp= &b;
    bp->b= 200;
    bp->show();
    return 0;
}
```

Output:

```
bp points to the base class object
b = 100
bp points to the derived class object
b= 200;
```

Explain about polymorphism. (Binding)

- It is a Greek word.
- Poly means many.
- Morph means forms.
- Polymorphism means "many forms"

After the inheritance it is another most important feature of OOP. In programing, polymorphism is the ability for objects of different classes related by inheritance to response differently to the same function call.

In polymorphism, the member functions with same name are defined in base class and also in each derived class.

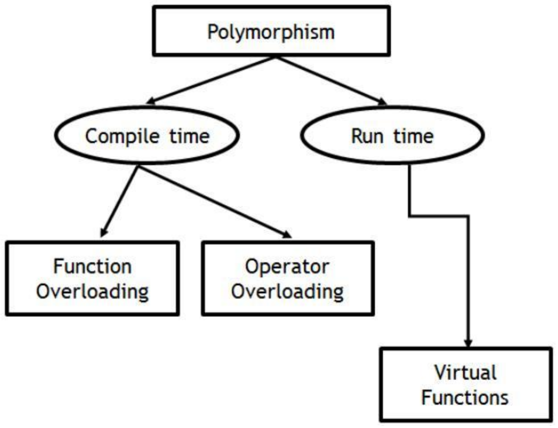


Fig. Types of polymorphism

A program in C++ is executed sequentially line by line. After compilation each line of source code is transformed in to machine language. When a function call is encountered by the compiler during execution, the function call is transformed into machine language and a sequential address is provided. Thus, *binding* refers to the process that is used for converting functions and variables into machine language. The C++ supports two types of binding. They are:

1. Static (or early) binding
2. Dynamic (or late) binding

Static binding

This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. So, it is called *early binding* or *compile time binding* or *static linking.*

This type of polymorphism is implemented using overloaded functions and operators. The overloaded member functions are selected for invoking by matching arguments, both type and numbers.

Example

```
#include<iostream.h>
class A
{
        int a;
  public:
     A( ) {      a = 1;      }
     void show( )                    //base class member function
     {    cout <<"\na="<<a;         }
};

class B: public A
{
        int b;
  public:
     B( ) {      b = 2;      }
     void show( )                    //derived class member function
     {    cout <<"\nb="<<b;         }
};
```

```
int main()
{
  A *bptr;
  A a; bptr=
  &a; bptr-
  >show();

  B b;
  bptr= &b;
  bptr->show();
  return 0;
}
```

Output:
a=1
a=1

Dynamic binding

The appropriate version of function will be invoked at runtime. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding.

This also known as dynamic binding, since the selection of appropriate function is done dynamically at runtime. It is implemented with virtual functions.

Example

```
#include<iostream.h>
class A
{
        int a;
  public:
     A( ) {      a = 1;     }
     virtual void show( )                //base class member function
     {    cout <<"\na="<<a;        }
};

class B: public A
{
        int b;
  public:
     B( ) {      b = 2;     }
     void show( )                //derived class member function
     {    cout <<"\nb="<<b;       }
};

int main()
{
  A *bptr; A a;
  bptr= &a;
  bptr->show();
```

```
   B b; bptr=
   &b; bptr-
   >show();
   return 0;
}
```

Output:
a=1
b=2

Explain about base and derived class objects.

In inheritance the derived class inherits all the properties of the base class. So pointers to the base class objects are type compatible with the pointers to derived class objects. So a base class pointer can point to the object of the derived class, but a pointer to the derived class object cannot point to the object of the base class.

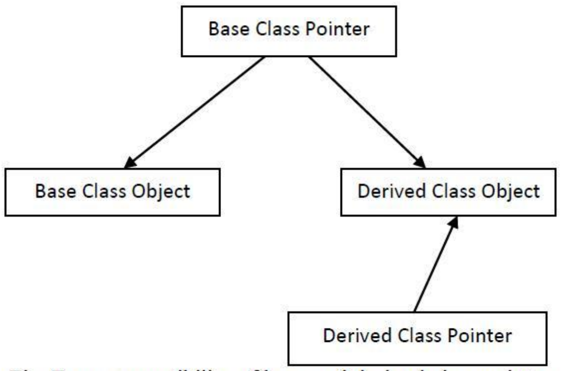


Fig. Type compatibility of base and derived class pointers

Example

```
# include
<iostream.h> class A
{
          protected: int
          a; public:
                    A(int x)
                    { a=x; } void
                    show()
                    {cout<<"\nIn base class";
                              cout<<"\na="<<a;
                    }
};

class B : public A
{
protected: int b;
public:
B(int x, int y) :A(x)
{         b=y;                }
void show()
{
cout<<"\n In derived class B";
cout<<"\na="<<a;
cout<<"\nb="<<b;
}
};

int main()
{
        A *bptr;
        bptr= new A(20);            //pointer to class A
        bptr->show();
        delete bptr;
        bptr = new B(10,20);        //pointer to class B
        bptr->show();
        ((B*)bptr)->show();
        return 0;
}
```

Output

```
In base class A
a=20
In base class A
a=10
In derived class B
a=10
b=20
```

What is a virtual function? What are its rules? Explain with an example.

(OR)

Explain about runtime polymorphism with an example.

In inheritance the same function name in both base and derived classes, the function in base class is declared as virtual using virtual keyword precedes its normal declaration. When a function is made *virtual*, C++ determines which function is to be used at run time based on the type of object pointed by the base pointer, rather than the type of the pointer.

Rules:

1. The virtual functions must be members of some class.
2. They can"t be static members.
3. They are accessed by using object pointers.
4. A virtual function can be friend of another class.
5. Constructors can"t be *virtual*. Destructors can be declared as virtual.
6. The prototypes of base class and derived class must be same; otherwise the virtual mechanism is ignored.
7. The operator overloading is also supports the virtual mechanism.
8. If a base class contains virtual function and if the same function is not redefined in derived classes, in such a case, the base class function is invoked.

Ex:

```
# include <iostream.h>
class Base
{
        public:
            void disp()
            {
                  cout<<"\nBase class disp() function";
            }
            virtual void show()
            {
                  cout<<"\n\nBase class show() function";
            }
};

class Derived: public Base
{
         public:
            void disp()
            {
                  cout<<"\nDerived class disp() function";
            }
            void show()
            {
                  cout<<"\n \nDerived class show() function";
            }
};
int main()
{
         Base b;
         Derived d;
         Base *bptr;
         bptr=&b;
         bptr->disp( );
         bptr->show( );
         bptr=&d;
         bptr->disp( );
         bptr->show( );
        return 0;
}
```

Output

```
Base class disp() function
Base class show() function
Base class disp() function
Derived class show() function
```

What is a pure virtual function? What is its use?

Pure virtual functions are also called do-nothing functions. These are always virtual functions. Virtual functions are defined with a null body and they may be overridden by derived classes. Pure virtual functions are defined as follows.

virtual void disp( )= 0;

After declaration of a pure virtual function in a class, the class becomes **abstract class** and are not able to declare an object. The classes derived from the pure abstract classes are required to re-declare the pure virtual function. All other derived classes without pure virtual functions are called **concrete classes.** These classes can be used to create the objects.

Example

```
# include <iostream.h>
class first
{
        protected: int b;
        public:
                first( ){          b=10;          }
                virtual void disp()=0;          //pure virtual function
};
class second: public first
{
        int d;
        public:
                second( ) {     d=20;          }
                void disp( )
                {
                        cout<<"\n b="<<b<<"\t d="<<d;
                }
};

void main( )
{
        first *p;
        second s;
        p=&s;
        p->disp();
}
```

Output:

```
b=10 d=20
```

What is abstract class? When do we use it?

Sometimes implementation of all function cannot be provided in a base class because we don‟t know the implementation. Such a class is called abstract class.

- The base class that has one or more pure virtual function is called the abstract base class.
- These classes designed as a framework or layout for derived classes.
- We cannot create objects of abstract classes.
- An abstract base class cannot be instantiated i.e. an object of its type cannot be defined but a pointer to an abstract base class can be defined.

For example, let *Shape* be a base class. We cannot provide implementation of function *area( )* in *Shape*, but we know every derived class must have implementation of ***area( ).***

Example:

```
#include <iostream.h>
class Shape                    // Abstract class
{
   protected:
      float l;
   public:
      void getData()
      {
         cin >> l;
      }
      virtual float area() = 0;                    // virtual Function
};

class Square : public Shape
{
   public:
      float area()
      {  return l*l;     }
};

class Circle : public Shape
{
   public:
      float area()
      { return 3.14*l*l; }
};

int main()
{
   Square s;
   Circle c;
   cout << "Enter the value of side: ";
   s.getData();
   cout<<"Area of square: " << s.area();
   cout<<"\nEnter radius of a circle: ";
   c.getData();
   cout << "Area of circle: " << c.area();
   return 0;
}
```

Output

```
Enter the value of side: 4
Area of square: 16
Enter radius of a circle: 2
Area of circle: 12.56
```

Write a program for illustrating working of virtual functions.

(OR)

How dynamic binding can takes place? Explain.

(OR)

Write about VTABLE and VPTR.

Binding means linking the function call and real function. Binding can be done either *compile time* or *runtime*. *virtual* keyword prevents the compiler from performing compile time binding and the binding is postponed until program execution.

Example

```
# include
<iostream.h> class A
{
        private : int i;
        public: virtual void show()
        {
                cout<<"\n In class A";
        }
};

class B
{
        private : int i;
        public: void show()
        {
                cout<<"\n In class B"
        }
};

class C
{
        public: void show()
        {
                cout<<"\n in class C";
        }
};
int main()
{
A a;
B b;
C c;
cout<<"\n Size of a="<<sizeof(a);
cout<<"\n Size of b="<<sizeof(b);
cout<<"n Size of c="<<sizeof(c);
}
```

Output:

```
Size of a = 4
Size of b = 2
Size of c = 1
```

The function *show()* of class A is virtual. Without virtual the size is 2, 2, 1. The size of object x with a virtual function in class A is 2 bytes for data member and 2 bytes for void pointer. Whenever a function is defined as virtual the compiler inserts a void pointer. In C++ the object without any data member will occupies 1 byte because every object have an individual address. So, minimum size is considered. „1" is the minimum non-zero positive integer.

When an object of base or derived class is created, a void pointer is inserted in the VTABLE. The VPTR points to the VTABLE. VTABLE is used to perform late binding. For late binding the compiler establishes VTABLE for class and its derived classes having virtual functions.

If no function is redefined in the derived class that is defined as virtual in the base class, the compiler takes the address of the base class function. The VPTR should be initialized with the beginning address of the relevant VTABLE. It is used to determine the type of the object i.e. either base class object or derived class object.

What are virtual destructors? Give an example.

The virtual destructors are implemented same as virtual functions. In constructors and destructors hierarchy of base and derived classes are constructed and destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.

For example, a derived class object constructed using a new operator. The base class pointer object that holds the address of the derived class object. When a base class pointer is destructed using the delete operator, the destructor of the base and derived class is executed. The following example explains this:

```
#include <iostream.h>
#include <conio.h>
class B
{
        public:
          B( )
         {
                cout<<"\n In B constructor";
         }
         virtual ~B( )
         {
                cout<<"\n In B destructor";
        }
};

class D : public B
{
        public:
           D( )
          {
                cout<<"\n In D constructor";
          }
          ~D()
         {
                cout<<"\n In D destructor";
         }
};
int main()
{
        B *ptr;
        ptr=new D;
        delete ptr;
        return 0;
}
```

Output:

```
In B constructor
In D constructor
In D destructor
In B destructor
```