UNIT-IV

Pointers

A pointer is a variable which is used to store address of another variable.i.e pointer value is an address.
Syntax is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C++ type and var-name is the name of the pointer variable.
Every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory.

```
int   *ip;   // pointer to an integer
double *dp;   // pointer to a double
float  *fp;   // pointer to a float
char   *ch    // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Example-1:

```
#include <iostream>
using namespace std;
int main()
{
int number=30;
int *  p;
p=&number;//stores the address of number variable
cout<<"Address of number variable is:"<<&number<<endl;
cout<<"Address of p variable is:"<<p<<endl;
cout<<"Value of p variable is:"<<*p<<endl;
  return 0;
}
```

Example-2 : Swap two numbers using pointers

```
#include <iostream>
using namespace std;
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
  return 0;
}
```

Features of Pointers

- Pointers save memory space.
- Execution time with pointers is faster because data are manipulated with the address, that is, direct access tomemory location.
- Memory is accessed efficiently with the pointers. The pointer assigns and releases the memory as well. Hence it can be said the Memory of pointers is dynamically allocated.
- Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensionalarrays.
- An array, of any type can be accessed with the help of pointers, without considering its subscript range.
- Pointers are used for file handling.
- Pointers are used to allocate memory dynamically.
- In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.

Pointer to class

To access members of a class using pointer we have to use the member access operator -> operator.

We can **define pointer of class** type, which can be used to point to class objects.

Example-1:

```
class Simple
{
   public:
   int a=5;
};
int main()
{
   Simple obj;
   Simple* ptr;   // Pointer of class type
ptr = &obj;

cout<<obj.a;
cout<<ptr->a;  // Accessing member with pointer
}
```

Example-2

```
#include <iostream>
using namespace std;
class Box {
private:
    double length;
    double breadth;
    double height;
```

```
  public:
    // Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0) {
cout<<"Constructor called." <<endl;
      length = l;
      breadth = b;
      height = h;
    }
    double Volume() {
      return length * breadth * height;
    }

};

int main() {
  Box Box1(3.3, 1.2, 1.5);   // Declare box1
  Box Box2(8.5, 6.0, 2.0);   // Declare box2
  Box *ptrBox;               // Declare pointer to a class.

  // Save the address of first object
ptrBox = &Box1;

  // Now try to access a member using member access operator
cout<< "Volume of Box1: " <<ptrBox->Volume() <<endl;

  // Save the address of second object
ptrBox = &Box2;

  // Now try to access a member using member access operator
cout<< "Volume of Box2: " <<ptrBox->Volume() <<endl;

  return 0;
}
```

Pointer Objects

When an object is a pointer to the class, the member functions of that class is accessed by using an object of that class with an arrow (->) operator.

Let us consider an example:

```
#include <iostream>
using namespace std;
class Sample { // This a class Sampleprivate:
            int value1, value2;
      public:
            void setValues(int num1, int num2) {
                  value1 = num1;
                  value2 = num2;
            }
            void display() {
                  cout<< "The given values are : " << value1 << " " << value2 <<endl;
            }};
```

```
int main() {
        Sample *p = new Sample; // A pointer object  p is created and stored the address of the
//dynamically allocated memory of a class Sample
        p->setValues(88, 99); // Pointer object p can access the public member functions using ->//operator
        p->display();
}
```

In the above example, a pointer object is referenced to the class Sample and the pointer object can access its public members by using arrow (->) operator.

Pointers to base class

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.

```
#include <iostream>
using namespace std;
class Polygon {
 protected:
  int width, height;
 public:
  void set_values (int a, int b)
 { width=a; height=b; }
};

class Rectangle: public Polygon {
 public:
  int area()
 { return width*height; }
};

class Triangle: public Polygon {
 public:
  int area()
 { return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * p1 = &rect;
  Polygon * p2 = &trgl;
  p1->set_values (4,5);
  p2->set_values (4,5);
cout<<rect.area() << '\n';
cout<<trgl.area() << '\n';
  return 0;
}
Output:
20
10
```

Function main declares two pointers to Polygon (named p1 and p2). These are assigned the addresses of rect and trgl, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon.

Dereferencing p1 and p2 (with p1-> and p2->) is valid and allows us to access the members of their pointed objects. For example, the following two statements would be equivalent in the previous example:

```
p1->set_values (4,5);
rect.set_values (4,5);
```

But because the type of both p1 and p2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and not those of the derived classes Rectangle and Triangle. That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members.

Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class.

Pointer to Derived Class

In C++, we can declare a pointer points to the base class as well as derive class.
Consider below example to understand pointer to derived class.

```
#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;
bptr=&b;        //address of base class
cout<<"Pointer of base class points to it";
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();  //executes base show only
      derive *d1;//derived pointer
      d1=&d;
    // d1=&b;// error: invalid conversion from 'base*' to 'derive*'
    cout<<"\npointer to derived class";
      d1->n1=100;
      d1->n2=200;
      d1->show();
      return 0;
    }
```

Output

```
Pointer of base class points to it
n1 = 44
n1 = 66
pointer to derived class
n1 =100
n2 =200
```

Here the show() method is the overridden method, bptr execute show() method of 'base' class twice and display its content. Even though bptr first points to 'base' and second time points to 'derive' ,both the time bptr->show() executes the 'base' method show().
If we create pointer for derived class and assign derived class object then we access derived class members.

this pointer

First we have to know how objects look at functions and data members of a class is 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.

Every object in C++ has access to its own address through an important pointer called this pointer.

Uses of this:

1) When local variable's name is same as member's name
2) To return reference to the calling object

When local variable's name is same as member's name

```
#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;
}
```

To return reference to the calling object

```
#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;
}
```

Polymorphism

Polymorphism is derived from two Greek words poly and morph. The word poly means many and morph means form. So polymorphism means many forms.

The process of representing one form in multiple forms is known as Polymorphism.

In C++, polymorphism is divided into two types:

- Compile time polymorphism or Static binding or Early binding : This is achieved by function overloading or operator overloading
- Runtime_polymorphism or Dynamic binding or Late binding : This is achieved by function overriding

A binding refers to the process that is to be used for converting functions and variables into machine language addresses.

Binding means matching the function call with the correct function definition by the compiler. It takes place either at compile time or at runtime.

In **early binding**, the compiler matches the function call with the correct function definition at compile time. It is also known as static binding or compile time binding.

By default, the compiler goes to the function definition which has been called during compile time.

In **late binding**, the compiler matches the function call with the correct function definition at runtime. It is also known as dynamic binding or runtime binding.

In late binding, the compiler identifies the type of object at runtime and then matches the function call with the correct function definition.

By default, binding takes place early. Dynamic binding can be achieved by declaring virtual functions.

Virtual functions

In C++, when a derived class inherits from a base class, an object of the derived class may be referred to via a pointer of the base class type instead of the derived class type.

If there are base class methods overridden by the derived class, the method actually called by such a pointer can be bound either early (by the compiler), according to the declared type of the pointer or reference, or late (by the runtime), according to the actual type of the object referred to.

When a base and derived classes have the member functions with the same name, at the time of compilation the compiler does not know which function is to be executed for the function call.

To avoid such problems, user need to use the runtime polymorphism depending on the virtual functions by using the keyword virtual.

The virtual function is a member function that is declared within the base class and redefined in the derived classes.

If the function is **virtual** in the base class, the method is resolved late and the derived class implementation of the function is called according to the actual type of the object referred to, regardless of the declared type of the pointer.

If the function is not virtual, the method is resolved early and the function called is selected according to the declared type of the pointer.

The general format of defining virtual functions is:

```
class Base {
        public:
                virtual return-type functionName() {
                        ... // This function is virtual and same function is available in Derived
class
                }
};
class Derived : visibility-mode Base {
        public:
                return-type functionName() {
                        ...
                }
};
```

Let us consider an example:

```
#include <iostream>
using namespace std;
class Base { // This is class Base
public:
        virtual void display() { // This is virtual and tells the compiler that this function binding
        // is done only at runtime
                        cout<< "This is base class display()" <<endl;
                }
};
```

```
class Derived : public Base {  // This is class Derived derived from Base class publicly
        public:
                void display() {
                        cout<< "This is derived class display()" <<endl;
                }
};
int main() {
        Base ob1; // Object is created to the Base class
        Derived ob2; // Object is created to the Derived class
        Base *p; // Pointer object is created to the Base class
        p = &ob1; // Address of Base class object is stored in the pointer object
        p->display(); // Accessing display() method of Base class and it is valid
        p = &ob2; // Address of Derived class object is stored in the pointer object
        p->display(); // Accessing display() method of Derived class and it is also valid
}
```

In the above example, the display() method in Base class is made as virtual.

While accessing display() method through a pointer object it will call the Base class display() or Derived class display() at runtime depending on the address contained in pointer object.

Rules that must be followed when creating virtual functions are:

- Virtual functions should not be static but must be a member of the class.
- They should be defined in public section of the class.
- It is possible to define virtual functions outside the class, in this case, the declaration is done inside the class and the definition is outside the class. The keyword virtual should be placed in declaration part but not in definition part.
- It also possible to return a value from virtual functions.
- Virtual functions may be declared as a friend for another class.
- The prototype in a base class and derived class must be identical for the virtual function to work properly.
- A class cannot have virtual constructors, but can contain a virtual destructor.
- A virtual function is present in the base class but if the same name of the function is not redefined in the derived class then the base class function is invoked every time.
- It is possible to have virtual operator overloading.

Access Private functions using Virtual functions

A user can call private member function of the derived class from the base class pointer with the help of virtual keyword.

The compiler checks for access specifier only at compile time. So at runtime when late binding occurs, it does not check whether it is calling the private function or the public function.

Let us consider an example:

```
#include <iostream>
using namespace std;
class Base { // This is class Base

        public:
                virtual void display() { // This is virtual and tells the compiler that this function
binding is done only at runtime
                        cout<< "This is base class display()" <<endl;
                }
};
class Derived : public Base {  // This is class Derived derived from Base class publicly

        private:
                void display() { // This is a private method
                        cout<< "This is derived class display()" <<endl;
                }
};
int main() {
        Derived ob2; // Object is created to the Derived class
        Base *p; // Pointer object is created to the Base class
        p = &ob2; // Address of Derived class object is stored in the pointer object
        p->display(); // Late binding occurs and which can access the display() method of
Derived class
}
```

In the above example, the display() method in Base class is made as virtual.

The late binding is done at accessing display() method through a pointer object which contains the address of Derived class object. So, it does not check whether the display() is private or public.

Pure Virtual functions

A pure virtual function is a function that has no definition within the base class.

Pure virtual methods typically have a declaration (signature) and no definition (implementation).

A pure virtual function is declared in the base class and cannot be used for any operation.

The class which contains a pure virtual function cannot be used to declare objects, such classes are known as abstract classes.

If anyone attempts to declare an object to abstract classes then the compiler would show an error.

A pure virtual function or pure virtual method is a virtual function that is required to be implemented by a derived class if the derived class is not abstract.

All the derived classes without pure virtual functions are called as concrete classesi.e., they can be used to create objects.

The format of a pure virtual function is:

```
virtual return-type function-name() = 0;
```

In C++, a class is abstract if it has at least one pure virtual function.

Let us consider an example:

```
#include <iostream>
using namespace std;
class Base { // This is class Base
        public:
                virtual void display() = 0; // It is a pure virtual function
};
class Derived : public Base {  // This is class Derived derived from Base class publicly
        public:
                void display() { // It is the overridden method of pure virtual function
                        cout<< "This is derived class display()" <<endl;
                }
};
int main() {
        Derived ob; // Object is created to the Derived class
        Base *p; // Pointer object is created to the Base class
        p = &ob; // Address of Derived class object is stored in the pointer
        p->display(); // It will access the display() method of Derived class
}
```

In the above example, the display() method in Base class is made as pure virtual function.

While accessing display() method through a pointer object it will call the Derived class display() at runtime.

Abstract classes

In C++, a class is abstract if it has at least one pure virtual function.

The class which contains a pure virtual function cannot be used to declare objects, such classes are known as abstract classes.

Abstract classes are used to provide an interface for its subclasses. If the derived class does not override the pure virtual function of the base class, then the derived class also becomes an abstract class.

Abstract classes cannot be instantiated but the pointers and references of abstract class type can be created.

An abstract class can have constructors.

```
#include <iostream>
using namespace std;
class Base {
        protected:
                int value;
                public:
                        Base(int number) {
                                value = number;
                        }
};                      virtual void display() = 0;
class Derived : public Base {
        public:
                Derived(int number) : Base(number) {

                }
                void display() {
                        cout<< "The given value : " << value <<endl;
                }
};
int main() {
        Base *p;
        Derived ob(55);
        p = &ob;
        p->display();
}
```

Virtual Destructors

Need of virtual Destructor

A destructor is used to destroy the objects that have been created by a constructor.

A destructor cleans up the storage (memory area of the object) that is no longer accessible.

The compiler automatically invokes the destructor when the object goes out of scope.

Destructors can also be used in inheritance concept. Destructors are invoked in the order opposite to the order in which constructors are called.

While using the polymorphism concept, deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour.

For example, the following program results in undefined behaviour:

```
#include <iostream>
using namespace std;
class Base { // This is class Base

        public:
                Base() {
                        cout<< "This is base class constructor\n";
                }
                ~Base() {
                        cout<< "This is base class destructor\n";
                }
};
class Derived : public Base {  // This is class Derived derived from Base class publicly

        public:
                Derived() {
                        cout<< "This is derived class constructor\n";
                }
                ~Derived() {
                        cout<< "This is derived class destructor\n";
                }

};
int main() {
        Base *p = new Derived; // Base class pointer pointing anonymous object of Derived class
        delete p; // Deleting the memory pointed by p
}
```

In the above example, delete p will only call the Base class destructor, which is undesirable because, the object of Derived class remains unrestricted because its destructor is never called which results in memory leak.

A constructor cannot be virtual because the constructors are always called in an order of base constructor and derived constructor respectively.

So, there is no necessity of making constructor as virtual.

Virtual Destructors

The destructors can be declared as virtual because in polymorphism the correct execution of destructors is only done by using virtual for base class destructors.

Destructors of the base and derived classes are called when a derived class object address pointed by a base class pointer object is deleted.

Let us consider an example:

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

```
class Base { // This is class Base

        public:
                Base() {
                        cout<< "This is base class constructor\n";
                }
                virtual ~Base() {
                        cout<< "This is base class destructor\n";
                }
};
class Derived : public Base {  // This is class Derived derived from Base class publicly

        public:
                Derived() {
                        cout<< "This is derived class constructor\n";
                }
                ~Derived() {
                        cout<< "This is derived class destructor\n";
                }

};
int main() {
        Base *p = new Derived;
        delete p;
}
```

In the above example, *p is a pointer object of the class Base.

The new operator allocates dynamic memory to the class Derived and then the anonymous object address is assigned to the Base class pointer p.

When the memory is allocated to the object of class Derived, it calls the constructors of Base and Derived classes respectively.

While deleting the memory it calls destructors, they are called in the order of Derived class destructors and Base class destructors only by placing virtual at the Base class destructor.

Inline functions

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called.

With inline keyword, the compiler replaces the function call statement with the function code itself and then compiles the entire code.

Thus with inline functions, the compiler does not have to jump to another location to execute the function and then jump back as the code of the called function is already available to the calling program.

If a function is big in terms of executable instructions then compiler can ignore the inline request and treat the function as a normal function.

The syntax for defining the function as inline is:

```
inline return-type function-name(parameters) {
// function code
}
```

```
include <iostream>
using namespace std;
inline int cube(int s) {
return s * s * s;
}

int main() {
int num;
cout<< "Enter a number : ";
cin>> num;
cout<< "The cube of a given number : " << cube(num) << "\n";
}
```

Static Data Members

The data members and member functions of a class may be qualified as static, so there may be static data members and static member functions.

A static data member of a class is just like a global variable for its class, i.e. the static data member is globally available for all the objects of that class type.

The static data members are usually maintained to store values common to the entire class.

For instance, a class may have a static data member keeping track of its number of existing objects.

A static data member has the following properties:

- It is initialized to zero when the first object of its class is created and no other initialization is permitted to the same static data member.
- Only one copy of the static data member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.
- It is visible only within the class but its lifetime is the entire program.

The declaration of a static data member within the class definition is similar to any other variable declaration except that it starts with the keyword static as shown below:

Syntax:

```
static data_typemember_name;
```

Example:

```
class Sample {
        static int count;
        ...
        ...
};
```

The above declared static data member count can be defined outside the class as:

Syntax: data_typeclass_name :: member_name =value;

Example:

```
int Sample::count;
```

Note that the type and scope of each static data member must be defined outside the class which is necessary because the static data members are stored separately rather than a part of an object.

Since they are associated with the class rather than an object, they are also known as class variables.

Static variables are initialized to zero by default. A user can also initialize a value to the static variable as:

```
int Sample::count = 10;
```

```
#include <iostream>
using namespace std;
class StaticData {
        char name[25];
        int id;
        static int count;
        public :
                void getData() {
                        cout<< "Enter student id and name : ";
                        cin>> id >> name;
                        count++;
                }
                void getCount() {
                        cout<< "Objects count is : " << count <<endl;
                }
                void putData() {
                        cout<< "Student id : " << id << " name : " << name <<endl;
                }
};
int StaticData::count;
int main() {
        StaticData s1;
        s1.getData();
        s1.getCount();
```

```
    StaticData s2, s3;
    s2.getData();
    s3.getData();
    s1.putData();
    s2.putData();
    s3.putData();
    s3.getCount();
}
```

Static Member Functions

A static member function can access only the static data members of the same class and it does not access any non-static data members.

A static member function can be declared in the class definition by using the keyword static as:

```
static return-type function-name(arguments) {
    ...
}
```

A static member function can be called by using class name instead of object as:

```
class-name::function-name();
```

```
#include <iostream>
using namespace std;
class Item {
        int itemNum;
        float price;
        static int count;
        public :
                void getData() {
                        cout<< "Enter item number and price : ";
                        cin>>itemNum>> price;
                        count++;
                }
                static void getCount() {
                        cout<< "Objects count is : " << count <<endl;
                }
                void putData() {
                        cout<< "Item number : " <<itemNum<< " price : " << price <<endl;
                }
};
int Item::count;
int main() {
        Item i1;
        i1.getData();
        Item::getCount();
        Item i2, i3;
        i2.getData();
        i3.getData();
        i1.putData();
        i2.putData();
```

```
    i3.putData();
    Item::getCount();
}
```

Static Objects

The static keyword can be applied to local variables, member functions, data members and as well as objects in C++.

An object becomes static when the static keyword is used in its declaration.

The static objects are initialized only once and destroyed only when the entire program terminates.

The static objects are active only within their scope but they are alive throughout the program.

In static objects, each data member is initialized to zero by default which does not happen to normal objects.

```
#include <iostream>
using namespace std;
class Sample {
        int num1, num2;
        public :
                void add() {
                        num1 += 5;
                        num2 += 9;
                }
                void show() {
                        cout<< num1 << " " << num2 <<endl;
                }
};
int main() {
        static Sample s;
        cout<< "Before addition : ";
        s.show();
        s.add();
        cout<< "After addition : ";
        s.show();
}
```