# Unit-5

# POINTERS, VIRTUAL FUNCTION AND POLYMORPHISM

# SYLLABUS

5.1 Pointers to objects

5.2 Develop programs using pointers to objects

5.3 ‘this’ pointer

5.4 Pointer to derived class

5.5 Virtual functions

5.6 Pointer to Virtual function

# 5.1 Pointers to objects

- Pointer is a variable which stores the address of another variable.
- We can also call the class members using the pointer.
- For that we have to create a pointer of the class data-type.
- Object pointers are useful in creating objects at run time.
- We can use the object pointer to access the public members of an object.

# Example:

```
Class person
{
----
  public:
void getdata( )
{
   ----
}
void putdata( )
{
----
}
};
```

In the given example, we can call the member function getdata( ) and putdata( ) using the object name as well as the using the object pointer.

```
person p1;          // create person object
person *p;          //create person to pointer
object
p=&p1;  // initialize pointer with address of
object
p->getdata( );      // using  object pointer
p->putdata( );      // using  object pointer
p1.getdata( );      // using  object name
p1.putdata( );      // using  object name
```

we can also use the following method:

```
(*p).getdata( );
(*p).putdata( );
```

# 5.2 Develop programs using pointers to objects

```
#include<iostream.h>
#include<conio.h>
Class person
{
    private:
            char name[10];
            int age;
public:
void getdata( )
{   cout<<”Enter  Name and Age :”
    cin>>Name;
    cin>>age;
}
void putdata( )
{   cout<<”Name:”<<name;
    cout<<”Age:”<<age;
}
};
```

```
void main( )
{
person  p1;
person  *p;          //pointer to object
 p=&p1;
p->getdata( );     // using  object pointer
p->putdata( );       // using  object pointer

person  p2;
p2.getdata( );       // using  object name
p2.putdata( );       // using  object name
 p=new person;
p->getdata( );
p->putdata( );
}
```

**Output:**

```
Enter  Name and Age : Sunita   20
Name: Sunita
Age: 20
Enter  Name and Age : Anita    18
Name: Anita
Age: 18
```

# 5.3 ‘THIS’ POINTER

"A **this** pointer is automatically passed to a function when it is called.“

❖ **Applications:**

- this pointer is used to represent an object that invokes a member function.
- Access data member with this pointer like, **this->x=50;**
- this pointer is to return the object it points to like **return *this;**

# Example: Find the address of object of which it is member of class.

```
class test
     {
          private: int x;
          public: void show()
          {
            cout<<"My object's address= "<<this;
          }
   };
   void main()
   {
          test b1,b2;
          b1.show();
          b2.show();
   }
```

**output:**

```
          My object's address=0x7f4effec
          My object's address=0x7f4effed
```

# 5.4 Pointer to derived class

- Using the pointer of the base class, we can access only those members which are inherited from base class and not of the members of derived class.
- To access the members of derived class we have to use the pointer to the derived type.

**Example**:

```
#include<iostream.h>
#include<conio.h>
class Base
{   public:
          void show( )
          {         cout<<"I am Base class";
          }
};
class Derv :  public Base
{   public:
          void show( )
          {         cout<<"I am Derived class";
          }
};
void main()
{   Derv d, *dptr;
    dptr = &d;
    dptr ->show();
}
```

# 5.5 Virtual functions

**Polymorphism**:

- **Definition**: "It is an ability to take more than one form.“
- There are two types of polymorphism are available.
  - **Compile Time Polymorphism**
    - Operator Overloading
    - Function Overloading
  - **Run Time Polymorphism**
    - Virtual Function

- **Compile Time Polymorphism:** The procedure of function or operator overloading is done at the time of compilation (Early as during compilation) so it is known as compile time polymorphism.
- It is known as **Early binding** or **Static binding**

- **Run Time Polymorphism:** The selection of appropriate function are called at run time ( Late as after compile time ) so it is known as run time polymorphism.
- It is known as **Late binding** or **Dynamic binding**

# VIRTUAL FUNCTION

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

❖ **Requirements:**

- It must be member of some class.
- It can not be a static member.
- It can only accessed by pointers.
- It can be friend of another class.
- It must be defined in the base class.
- There is no virtual constructors, but we have virtual destructor.

# Example:

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:
virtual void show()
{
cout<<"I am Base class"<<endl;
}
};
class Derv1 : public Base
{
public:
void show()
{
cout<<"I am Derived 1class"<<endl;
}
};
```

```
class Derv2 : public Base
{
        public:
        void show()
        {
        cout<<"I am Derived 2 class"<<en
        }
};

void main()
{
Base  *bprt;
 int ch;
cout<<"\n 1. call function of  Derv1 class";
cout<<"\n 1. call function of  Derv2 class";
cout<<"\n Enter your choise";
cin>>ch;
        if(ch = =1)
        bptr  = new derv1;
        elseif(ch == 2)
                bptr  =  new derv2;
        bptr->show();
}
```

# PURE VIRTUAL FUNCTION

- In the virtual function, the function is declared as virtual inside the base class and redefine it in the derived classes,
- The function inside the base class is used for performing any task.
- It only serves as a placeholder.
- It is also called as a “do-nothing” function.
- It may be defined as follows: **virtual void show( )=0;**
- A pure virtual function is a function declared in a base class that has no definition relative to the base class.
- Example:

```
class Base
{       public:
        virtual void show() = 0   //it is pure virtual funciton

};
```

# 5.6 Pointer to Virtual function

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:
virtual void show()
{
cout<<"I am Base class"<<endl;
}
};
class Derv : public Base
{
public:
void show()
{
cout<<"I am Derived class"<<endl;
}
};
```

```
void main()
{
Base  bp;
Derv dv;
Base *bptr;
bptr=& bp;
bptr->show();
Bptr=& dv;
Bptr->show();
}
```

**Output:**

```
I am Base class
I am Derived class
```

In previous example the function call bptr->show() executes the function that corresponds to the contents of the pointer bptr, and not on the type of pointer.