Pointer to object (Object Pointer): C++ allows us to have pointer to objects. The pointers pointing to objects are known as object pointers. These pointers are declared by putting (*) in front of an object pointer's name.

Syntax for object-pointer declaration:

```
Class_name *object-pointer-name;
```

For example: Student *ptr;
Here, 'Student' is an already defined class-name. And 'ptr' is an object pointer that points to an object of type 'Student' class.

Example for object-pointer initialization:

```
Student s1;          //creating object 's1' of class 'Student'
Student *ptr = &s1;  // object pointer 'ptr' pointing to object 's1'
```

- The object pointer can be used to access the members of a class. The arrow operator (->) is used for this purpose instead of dot(.) operator.

For example:

```
ptr -> display( );
```

It calls the 'display' member function of class 'Student'.

To use the dot operator (.), we have to dereference the object pointer.
For example: (*ptr).display( );

An example program that demonstrates the use of object pointer is given below. *[Example program- 3.1]*

this pointer: A special pointer that points to the current object of the class with the help of 'this' keyword is known as 'this pointer'. The 'this' pointer is passed as a hidden argument to all the non-static member function calls. It is available as a local variable within their body and may be used to refer to the invoking object. It is not available in static member functions and friend functions.

Examples of use cases:

1. It can be used to know the address of the current object inside the called member function. *[Example program- 3.2]*

2. It can be used for name conflict resolution. That means, when the name of the local variable and data members is the same, 'this' pointer is used to distinguish between them. *[Example program- 3.3]*
3. It is used to return the object or reference of the object it is pointing to.

Example program: 3.1

```
class Russia {
  public:
   void declareWar(){
    cout<<"Attack on Ukraine."<<endl;
   }
   void goAhead(){
    cout<<"We are coming."<<endl;
   }
};
int main(){
  Russia r;  //Creating r object
  Russia *ptr; //Declaring an object pointer
  ptr = &r; //Initialing the object pointer
  ptr->declareWar(); //Calling member function via object pointer
 (*ptr).goAhead(); //Alternative way to call member function
                   //from object pointer
  return 0;
}
```

Output: Attack on Ukraine.
We are coming.

Example program: 3.2

```
class A{
 public:
     void fun(){
    //Here, 'this' pointer points to current object,i.e. obj1
      cout<<"Address of current object: "<<this<<endl;
     }
};

int main(){
  A obj1;
  obj1.fun(); //It implicitly sends 'this' pointer as hidden
              //argument to the fun() function.
  return 0;
}
```

Output: Address of current object: 0x61fe1f

Example program: 3.3

```
class A{
 int x,y;   //data members
 public:

 //names of local variables are same as that of data members
   void getData(int x, int y){
     /* x=x;  (name conflicts and so, values in local
        y=y;  variables cannot be assigned to the private
              members in such way) */

     this->x=x;    // 'this' pointer solves this problem
     this->y=y;
   }
   void display(){
    cout<<"The value of x: "<<x<<endl;
    cout<<"The value of y: "<<y<<endl;
   }
};

int main(){
  A obj1;
  obj1.getData(5,7); /*It implicitly sends 'this'
                       pointer as hidden argument to the
                       getData() member function. */
  obj1.display();
  return 0;
}
```

Output: The value of x: 5
The value of y: 7

Polymorphism: Polymorphism means having many forms. That means, the same entity (object, function, operator) behaves differently in different scenarios.

Polymorphism is a feature of OOP that allows the object, function or an operator to perform in different ways depending on how the object, function or operator is used.

Let us consider a real-life example of polymorphism. A person may behave as a student at a class-room, a son at home, and a player at playground. Here, the same person shows different behaviors in different situations. This is an example of a polymorphism.

Benefits/Advantages of polymorphism:

- Polymorphism allows us to reuse the codes, by creating functions and classes that can be operated for multiple use cases.
- It allows the same variable to store multiple data-types.
- It allows the same operators to be used for different scenarios.

Types of polymorphism: Polymorphism can be categorized into two types:
(a) Compile-time polymorphism and (b) Run-time polymorphism

(a) Compile-time polymorphism: It is a type of polymorphism in which an object is bound (destined) to its function at compile-time. In this type, the compiler decides the appropriate function to execute (the called function is chosen) for a particular function call at compile-time. The binding is done based on the type of object or a pointer.

The compile-time polymorphism is also known as early binding or static binding or **static polymorphism**. It can be achieved in two ways: function overloading and operator overloading.

(b) Run-time polymorphism: It is a type of polymorphism in which an object is bound (destined) to its function at run-time. In this type, the compiler decides the appropriate function to execute (the called function is chosen) for a particular function call dynamically at run-time. The binding is done based on the location pointed to by the pointer and not according to the type of object or a pointer.

The run-time polymorphism is also known as late binding, dynamic binding or **dynamic polymorphism**. It can be achieved with the help of function overriding with virtual functions.

Pointer to derived class: Pointers to object (object pointers) of base class are type-compatible with pointer to objects of the derived class. That means a pointer of the base class type can be used to point to the objects of base class as well as to its derived classes.

For example, if 'Parent' is a base class and 'Child' is the derived class from 'Parent', then a pointer declared as a pointer to 'Parent' can also be a pointer to 'Child'. It can be demonstrated by the following program example: *[Example program- 3.4]*

Example program: 3.4

```
class Parent{
  public:
  void display(){
   cout<<"From parent class"<<endl;
  }
};
class Child: public Parent{
  public:
  void display(){
    cout<<"From child class"<<endl;
  }
};

int main()
{
   Parent *ptr; //Object pointer of Parent class
   Parent p1;   //object of parent class
   Child c1;    //object of child class
   ptr =&c1;     //Object pointer of parent class pointing to child object
   ptr->display(); // It calls the function of parent class
   ((Child*) ptr)->display(); // Typecasting done. It calls the function of child class
    return 0;
}
```

Output: From parent class
From child class

In above program, the pointer 'ptr' of parent class is used to point to the child object 'c1'. It is valid with C++ because c1 is an object of class Child which is derived from the class Parent.

Note: Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. Only those members which are inherited from base class can be accessed and not the members that originally belong to the derived class. Another pointer that is declared as pointer to derived type can be used for this purpose.

Virtual function: It is a function that makes sure that the correct member function is called for an object, regardless of the type of reference used for the function call.

Virtual function is declared by using a keyword 'virtual' before the normal declaration of a member function in the base class. When a function is made virtual, the compiler decides which function to execute at run-time based on the type of object pointed by the pointer rather than the type of pointer. It tells the compiler to perform dynamic binding, and hence achieve **run-time polymorphism**.

When base class pointer contains the address of the derived class object, it always executes the base class function. In this case, the compiler ignores the contents of the (base) pointer and chooses the member function that matches the type of the pointer. This issue can only be resolved by using the 'virtual' 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. In this way **runtime polymorphism** is achieved.

Let us consider a following program example *[Example program- 3.5]*

Example program: 3.5

```
class A{
  public:
    void fun(){ // no virtual function
     cout<<"Fun from parent class A."<<endl;
    }
};
class B: public A{
  public:
    void fun(){
     cout<<"Fun from child class B."<<endl;
    }
};

int main()
{
  A *ptr; //object pointer of type base class A.
  B obj; //creating object of class B.
  ptr=&obj;
  ptr->fun();
  return 0;
}
```

Output: Fun from parent class A.

Explanation of the program: In the above program, an object pointer 'ptr' is created of type A which is a base class. It is used to point to the object 'obj' of derived class B. When this pointer calls the fun( ) function, the function of the base class is executed instead of the derived class due to early binding.

The solution to this problem is declaring the fun( ) function of base class as **virtual**. In the above program, if we make the function fun( ) a **virtual function** as follows:

```
class A{
  public:
    virtual void fun(){ // virtual function
     cout<<"Fun from parent class A."<<endl;
    }
};
```

Then, the output of the program would be: 'Fun from child class B'. This is due to late binding and it is the case of a **run-time or dynamic polymorphism**.

Pure virtual function: A do-nothing function that is only declared but doesn't have any definition is known as pure virtual function. It is achieved by appending '= 0' at the end of the declaration of a virtual function.

This function is declared in a base class and serves as a placeholder. Any child classes derived from it have to define the function or re-declare it as a pure virtual function.

For example:

```
class Book {
    public:
    virtual void display( )=0;    //pure virtual function
};
```

In the above class Book, the display( ) function is declared as a pure virtual function. This function is defined in the child classes derived from the class Book.

Abstract class: A class that consists of at least one pure virtual function is known as abstract class. Generally, an abstract class is designed to act as a base class.

An instance (object) of such class cannot be created. But, a pointer to an abstract class can be created, and this pointer can be used for selecting the proper functions.

For example:

```
class Book {
    public:
    virtual void display( )=0;    //pure virtual function
};
```

Here, Book has a pure virtual function. So, it is an **abstract class**. Any derived classes of it must override the function display( ) or they should declare it again as a pure virtual function. The function can be implemented in further inherited classes.

Q. Why is there the need of abstract class? (Uses of abstract class)
Solution: An abstract class is designed to act as a base class. Sometimes, implementation of all the functions cannot be provided in a base class. But these functions could be implemented in the child classes that are derived from the base class. For example, let 'Shape' be a base class and 'Circle' & 'Square' be its child classes. The Shape class has a function called draw( ). This function is not implemented in it but is implemented in its derived classes Circle and Square. So, in this type of situations, concept of abstract class is used.

An example program demonstrating the use of **abstract class** is given below *[Example program- 3.6]*

Difference between abstract class and concrete class:

Abstract Class	Concrete Class
1. A class that has at least one pure virtual function is known as abstract class.	1. An ordinary class that has no pure virtual function is known as concrete class.
2. An object (instance) of abstract class cannot be created.	2. An object (instance) of concrete class can be created.
3. It can have unimplemented methods.	3. All the methods have to be implemented.
4. In inheritance, it is typically a base class from which other classes are derived.	4. In inheritance, it is typically a derived class that implements all the missing functionalities.

5. Example:
class Shape {
public:
virtual void draw( )=0;
}
	5. Example:
	class Circle : public Shape {
	public:
	void draw( ){
	cout<<“Drawing circle.”;
	}
	};

Example program: 3.6

```
class Shape{     //Abstract class
 public:
   virtual void draw()=0;
};
class Circle: public Shape{
 public:
    void draw(){
     cout<<"Drawing circle."<<endl;
    }
};
class Square: public Shape{
 public:
    void draw(){
     cout<<"Drawing square."<<endl;
    }
};
```

```
int main()
{
  Shape *ptr;
  Circle c1;
  Square s1;
  ptr=&c1;
  ptr->draw();
  ptr=&s1;
  ptr->draw();
  return 0;
}
```

Output: Drawing circle.
Drawing square.

Virtual Destructors: It is a method which ensures that both the destructors of the base class and the derived class are called at runtime to prevent any unwanted memory leakage.

Suppose an object of derived class is created dynamically using the pointer of type base class.
For example: Base *ptr = new Derived;
where, 'Base' is a base class, and 'Derived' is a derived class.
When ptr is deleted, only the destructor of the base class is called, and the destructor of derived class is not called. This may lead to memory leaks in the program.

The solution for this is to use 'virtual destructor' in the base class by using a keyword 'virtual' before destructor definition.

Eg. virtual ~ Derived( ) {

}

Now, when ptr is deleted, both the destructors of derived class and the base classes are called to release the used up memory resources.

An example program that demonstrates the use of virtual destructor is given below. *[Example program- 3.7]*

Example program: 3.7

```
class A{
  public:
   ~A(){
     cout<<"Destructor from A"<<endl;
   }
};
class B: public A{
  public:
    ~B(){
      cout<<"Destructor from B"<<endl;
    }
};
int main()
{
   A* ptr;  //object pointer of type class 'A'
   ptr=new B;  //object of type class 'B' created dynamically
   delete ptr;  //releasing memory pointed by 'ptr'
   return 0;
}
```

Output: Destructor from A

Now in the above program, when the destructor of base class is made virtual, then both the destructors of class A and class B are called as shown below.

```
class A{
  public:
   virtual ~A(){    // Virtual destructor
     cout<<"Destructor from A"<<endl;
   }
};
class B: public A{
  public:
    ~B(){
      cout<<"Destructor from B"<<endl;
    }
};
int main()
{
   A* ptr;  //object pointer of type class 'A'
   ptr=new B;  //object of type class 'B' created dynamically
   delete ptr;  //releasing memory pointed by 'ptr'
   return 0;
}
```

Output: Destructor from B
Destructor from A

OLD /MODEL/IMPORTANT QUESTIONS:

1. **What is virtual function? When do we make a function virtual? Explain with appropriate example.**
2. **What is polymorphism? How can you achieve runtime polymorphism in C++? Discuss with a suitable example.**
3. **When do you use virtual function? How it provides run time polymorphism. Explain with a suitable example.**
4. **How can you achieve compile time and run time polymorphism? Explain with examples.**
5. **How can you define pure virtual functions in C++? The pure virtual functions do nothing but it is defined in base class, why?**
6. **Why is 'this' pointer widely used than object pointer? Write a program to implement pure polymorphism.**
7. **Discuss the role of this pointer and abstract class in object-oriented programming.**
8. **Can you derive a pointer from a base class? Explain with suitable example.**
9. **How do we make use of a virtual destructor when we need to make sure that the different destructors in an inheritance chain are called in order? Explain with an example in C++.**
10. **What are the advantages of runtime polymorphism over compile time polymorphism?**
11. **Differentiate between virtual function and pure virtual function.**