UNIT III – INHERITANCE AND POLYMORPHISM

Inheritance: Defining a class hierarchy, Different forms of inheritance, Defining the Base and Derived classes, Access to the base class members, Base and Derived class constructor, Virtual base class, **Virtual Functions and Polymorphism**, Static and Dynamic binding, Function overloading, virtual functions, Dynamic binding through virtual functions, Virtual function call mechanism, Pure virtual functions, Abstract classes.

CO3: Develop C++ programs using Inheritance and Polymorphism. (K3)

3.1 Inheritance: Defining a class hierarchy

In object-oriented programming, real-world entities are related to each other.

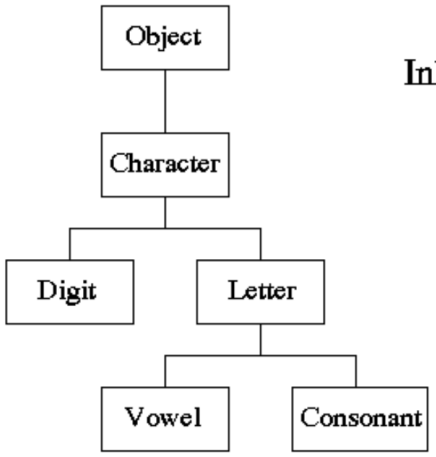
To represent these relationships in C++, we use Inheritance.

A class hierarchy is formed when:

- One class is used as a base
- Other classes are derived from it

This helps in organizing classes logically.

Example



Inheritance Tree

Class Hierarchy

A **class hierarchy** is a structure where:

- A **base class** is at the top
- One or more **derived classes** are connected below it
- Classes are arranged in **levels**

It shows **parent–child relationship between classes**.

Term	Explanation
Base Class	Class whose properties are inherited
Derived Class	Class that inherits from base class
Parent Class	Another name for base class
Child Class	Another name for derived class
IS-A Relationship	Derived class is a type of base class

Example: Student **is a** Person

Need of Class Hierarchy

Class hierarchy helps to:

- Represent **real-world structure**
- Reuse common data and functions
- Reduce duplication
- Improve clarity of program design
- Extend programs easily

Defining a Class Hierarchy

A class hierarchy is defined using **inheritance syntax**.

Syntax

```
class DerivedClass : access_specifier BaseClass {
    // members of derived class
};
```

Example

//Base Class

```
class Person {
public:
    int age;

    void setAge(int a) {
        age = a;
```

```
    }
};
```

//Derived Class

```
class Student : public Person {
public:
    int rollNo;

    void setRoll(int r) {
        rollNo = r;
    }
};
```

//using the Hierarchy

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

int main() {
    Student s;

    s.setAge(20);      // inherited from Person
    s.setRoll(101);    // own member

    cout << "Age: " << s.age << endl;
    cout << "Roll No: " << s.rollNo << endl;

    return 0;
}
```

How This Forms a Class Hierarchy?

- Person → Base class

- Student → Derived class
- Student object contains:
    - Data members of Person
    - Data members of Student

******

3.2 Different forms of Inheritance

Different forms of inheritance describe **how base and derived classes are connected**.

C++ supports the following forms:

1. Single Inheritance
2. Multilevel Inheritance
3. Multiple Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Each form represents a **different class hierarchy structure**.

1. Single Inheritance

Single inheritance involves:

- One base class
- One derived class

Structure

Base Class

↓

Derived Class

Example Program

```
class Person {
public:
    int age;
};
```

```
class Student : public Person {
public:
    int rollNo;
};
```

2. Multilevel Inheritance

In multilevel inheritance:

- A derived class becomes the **base class for another class**

Structure

Class A

↓

Class B

↓

Class C

Example Program

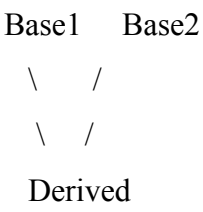
```
class A {
public:
    int x;
};

class B : public A {
public:
    int y;
};

class C : public B {
public:
    int z;
};
```

3. Multiple Inheritance

In multiple inheritance:

- A derived class inherits from **more than one base class**

Structure



Example Program

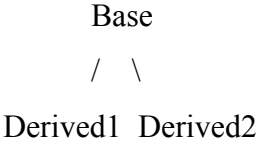
```
class A {
public:
    int x;
};

class B {
public:
    int y;
};

class C : public A, public B {
public:
    int z;
};
```

4. Hierarchical Inheritance

In hierarchical inheritance:

- One base class
- Multiple derived classes

Structure



Example Program

```
class Person {
public:
    int age;
};

class Student : public Person {
public:
    int rollNo;
};
class Teacher : public Person {
public:
    int empId;
};
```

5. Hybrid Inheritance

Hybrid inheritance is:

- A **combination of two or more inheritance forms**

Structure

```
    A
   / \
  B   C
   \ /
    D
```

- Combination of:
  - Multiple inheritance
  - Multilevel inheritance
- Can cause **diamond problem**

- ❖ Solved using **virtual base classes**

******

3.3 Defining the Base and Derived Class

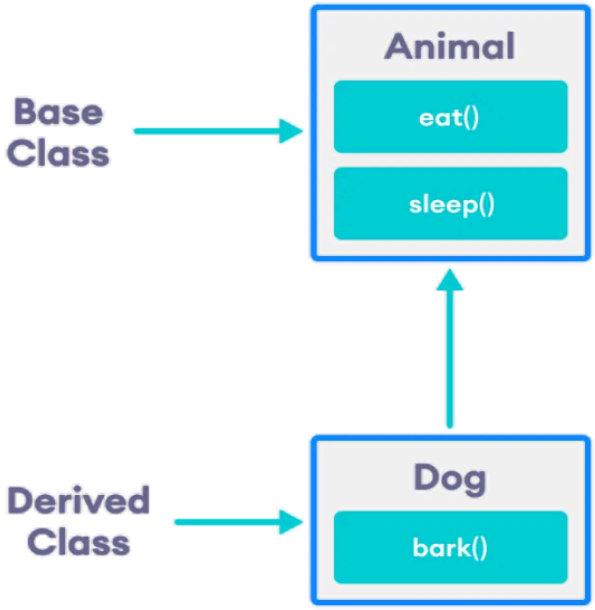
Base Class

A **base class** is:

- The **existing class**
- Contains **common data members and member functions**
- Used to derive other classes

It represents **general properties**.

Example




```
class Person {
public:
    int age;

    void setAge(int a) {
        age = a;
```

```
    }
};
```

- Person is the base class
- Contains common feature → age
- Can be reused by many derived classes

Derived Class

A **derived class** is:

- A **new class**
- Inherits features from a base class
- May add **new data and functions**

It represents **specialized properties**.

Example

```
class Student : public Person {
public:
    int rollNo;

    void setRoll(int r) {
        rollNo = r;
    }
};
```

- Student is derived from Person
- Inherits age
- Adds its own member rollNo

Syntax for Defining Base and Derived Classes

```
class DerivedClass : access_specifier BaseClass {
    // derived class members
};
```

Example Program

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

class Person {
public:
    int age;

    void setAge(int a) {
        age = a;
    }
};

class Student : public Person {
public:
    int rollNo;

    void setRoll(int r) {
        rollNo = r;
    }
};

int main() {
    Student s;

    s.setAge(20);     // base class function
    s.setRoll(101);   // derived class function

    cout << "Age: " << s.age << endl;
```

```
    cout << "Roll No: " << s.rollNo << endl;

    return 0;
}
```

Relationship Between Base and Derived Classes

Base Class (Person)

↓

Derived Class (Student)

Derived class object contains:

- Base class members
- Derived class members

Use of Base and Derived Class

- Avoid code repetition
- Improve program structure
- Support hierarchical design
- Easy to extend programs

******

3.4 Access to the Base Class Members

It describes how members of a base class are accessed in a derived class. It depends on access specifiers and type of inheritance.

C++ provides three access specifiers

Access Specifier	Meaning
public	Accessible everywhere
protected	Accessible in derived class
private	Accessible only inside the same class

Public Inheritance

Rule

Base class public → Derived class public

Base class protected → Derived class protected

Base class private → Not accessible

Example

```
class A {
public:
    int x;
protected:
    int y;
private:
    int z;
};

class B : public A {
public:
    void show() {
        x = 10;   // accessible
        y = 20;   // accessible
        // z = 30; // NOT accessible
    }
};
```

Protected Inheritance

Rule

Base class public → Derived class protected

Base class protected → Derived class protected

Base class private → Not accessible

Example

```
class A {
public:
```

```
    int x;
protected:
    int y;
};

class B : protected A {
public:
    void show() {
        x = 10;  // accessible as protected
        y = 20;  // accessible
    }
};
```

Private Inheritance

Rule

Base class public → Derived class private

Base class protected → Derived class private

Base class private → Not accessible

Example

```
class A {
public:
    int x;
protected:
    int y;
};
class B : private A {
public:
    void show() {
        x = 10;  // accessible as private
```

```
        y = 20;  // accessible
    }
};
```

Accessing Base Class Members using Object

```
int main() {
    B obj;
    obj.x = 10;  // only if public inheritance
}
```

Access depends on inheritance mode.

Comparison

Base Member	Public Inheritance	Protected Inheritance	Private Inheritance
public	public	protected	private
protected	protected	protected	private
private	Not accessible	Not accessible	Not accessible

******

3.5 Base and Derived class constructor

A **constructor** is:

- A special member function
- Same name as class
- Automatically invoked when an object is created
- Used to initialize data members

Constructors in Inheritance

When a derived class object is created, the base class constructor is called first, followed by the derived class constructor.

This happens **automatically**.

Example: Constructor Call Order

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

```
class Base {
public:
    Base() {
        cout << "Base class constructor called" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived class constructor called" << endl;
    }
};

int main() {
    Derived obj;
    return 0;
}
```

Output

```
Base class constructor called
Derived class constructor called
```

Why Base Class Constructor is Called First?

- Derived class object **contains base class part**
- Base class must be initialized before derived class
- Follows logical memory layout

Parameterized Base Class Constructor

If base class has a parameterized constructor, the derived class must pass arguments.

Example: Passing Parameters to Base Constructor

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

class Base {
public:
    Base(int x) {
        cout << "Base value: " << x << endl;
    }
};
class Derived : public Base {
public:
    Derived(int y) : Base(y) {
        cout << "Derived value: " << y << endl;
    }
};

int main() {
    Derived obj(10);
    return 0;
}
```

Output

```
Base value: 10
Derived value: 10
```

Syntax: Initializer List

```
Derived(int y) : Base(y) {
}
```

- Used to call base class constructor

- Mandatory for parameterized base constructors

Order of constructor execution in inheritance

When an object of a derived class is created, the base class constructor is executed first, followed by the derived class constructor.

Base and Derived class constructors

In inheritance, when a derived class object is created, the base class constructor is automatically invoked before the derived class constructor. This ensures proper initialization of base class members. If the base class has a parameterized constructor, the derived class must pass arguments using an initializer list.

******

3.6 Virtual base class

Problem in Multiple Inheritance

Consider this inheritance structure:

- Class D inherits from both B and C
- B and C both inherit from A

This creates two copies of base class A inside class D

This issue is called the Diamond Problem.

Why is the Diamond Problem an Issue?

- Ambiguity occurs while accessing base class members
- Compiler does not know which copy of base class to use
- Leads to confusion and errors

Virtual Base Class

A virtual base class is a base class that:

- Is shared by all derived classes
- Ensures only one copy of the base class exists
- Solves the diamond problem

Declaration of Virtual Base Class

Use the keyword virtual while inheriting the base class.

Syntax

```
class Derived : virtual public Base {
};
```

Example without Virtual Base Class (Problem Case)

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

class A {
public:
    int x;
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};

int main() {
    D obj;
    // obj.x = 10;  // ERROR: ambiguous
    return 0;
}
```

- D contains two copies of A
- Accessing x is ambiguous

Example using Virtual Base Class (Solution)

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

class A {
public:
    int x;
};

class B : virtual public A {
};

class C : virtual public A {
};

class D : public B, public C {
};

int main() {
    D obj;
    obj.x = 10;   // No ambiguity
    cout << obj.x << endl;
    return 0;
}
```

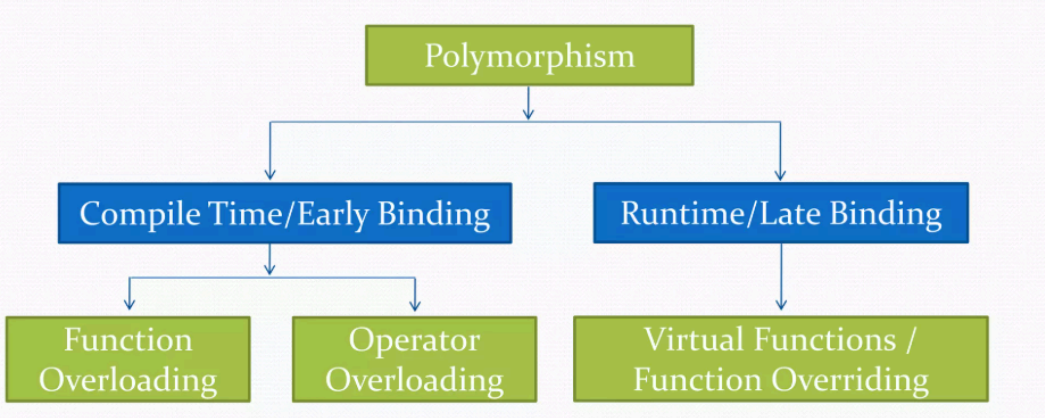
Virtual Base Class Works?

- A is declared as virtual
- Only one copy of A is shared
- Both B and C refer to the same base class A

******

3.7 Virtual Functions and Polymorphism

Virtual function belongs to the branch of Runtime Polymorphism



Polymorphism

Polymorphism means "many forms".

- The same function name
- Behaves differently
- Depending on the object type

One interface, many implementations.

Real-World Example

Shape → draw()

Circle → draw()

Rectangle → draw()

Same function name draw(), different behaviour

Types of Polymorphism

Type	Description
Compile-time Polymorphism	Function overloading
Run-time Polymorphism	Virtual functions

Virtual functions are used for run-time polymorphism

Virtual Function

A virtual function is:

- A member function of a class
- Declared using the keyword virtual
- Called based on the object type, not pointer type

Function call is resolved at run time.

Why Virtual Functions are Needed?

Without virtual functions:

- Base class pointer calls base class function
- Even if it points to derived object

With virtual functions:

- Correct derived class function is called

Example Program WITHOUT Virtual Function (Problem Case)

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

class Base {
public:
    void show() {
        cout << "This is Base class show()" << endl;
    }
};
```

```
class Derived : public Base {
public:
    void show() {
        cout << "This is Derived class show()" << endl;
    }
};

int main() {
    Base* b;
    Derived d;

    b = &d;
    b->show();   // Base class function called

    return 0;
}
```

Output

```
This is Base class show()
```

- This is NOT Polymorphism

Example Program WITH Virtual Function (Solution)

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

class Base {
public:
    virtual void show() {
        cout << "This is Base class show()" << endl;
    }
```

```
};

class Derived : public Base {
public:
    void show() {
        cout << "This is Derived class show()" << endl;
    }
};

int main() {
    Base* b;
    Derived d;

    b = &d;
    b->show();   // Derived class function called

    return 0;
}
```

Output

```
This is Derived class show()
```

How Virtual Function Works?

- Base class pointer → derived class object
- Function call resolved at run time
- Based on actual object type

This is called dynamic binding.

Static Binding vs Dynamic Binding

Binding Type	When decided	Example
Static Binding	Compile time	Normal functions
Dynamic Binding	Run time	Virtual functions

Conditions for Runtime Polymorphism

To achieve runtime polymorphism:

1. Inheritance must be used
2. Function must be declared **virtual**
3. Base class pointer must point to derived object

******

3.8 Static and Dynamic Binding

Binding

Binding is the process of connecting a function call to the actual function code.

Example

obj.show();

Which show() function should be executed?

This decision is called **binding**.

Types of Binding

C++ supports **two types of binding**:

1. Static Binding (Early Binding)
2. Dynamic Binding (Late Binding)

Static Binding (Early Binding)

Static binding is:

- Function call resolved at compile time
- Also called early binding

Characteristics

- Faster execution
- No runtime overhead
- Function call decided by **pointer type**

Example: Static Binding

```
#include <iostream>
```

```
using namespace std;

class Base {
public:
    void show() {
        cout << "Base class show()" << endl;
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "Derived class show()" << endl;
    }
};

int main() {
    Base* b;
    Derived d;

    b = &d;
    b->show();   // Static binding

    return 0;
}
```

Output

```
Base class show()
```

Why is this Static Binding?

- Function is not virtual
- Compiler binds function call at compile time
- Based on pointer type (Base*)

Dynamic Binding (Late Binding)

Dynamic binding is:

- Function call resolved at runtime
- Also called late binding
- Achieved using virtual functions

Characteristics

- Supports runtime polymorphism
- Function call decided by **object type**
- Slight runtime overhead

Example: Dynamic Binding

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

class Base {
public:
    virtual void show() {
        cout << "Base class show()" << endl;
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "Derived class show()" << endl;
    }
```

```
};

int main() {
    Base* b;
    Derived d;

    b = &d;
    b->show();   // Dynamic binding

    return 0;
}
```

Output

```
Derived class show()
```

Static vs Dynamic Binding

Feature	Static Binding	Dynamic Binding
Binding time	Compile time	Run time
Also called	Early binding	Late binding
Uses virtual	No	Yes
Function call based on	Pointer type	Object type
Supports polymorphism	No	Yes

When to Use Which?

- **Static binding**:
  - Normal functions
  - Function overloading
- **Dynamic binding**:
  - Runtime polymorphism
  - Base pointer → derived object

******

3.9 Function Overloading

Function overloading means: Defining multiple functions with the same name but with different parameter lists. The compiler decides which function to call at compile time.

Why Function Overloading is Needed?

- Improves code readability
- Uses same function name for similar operations
- Reduces number of function names
- Supports compile-time polymorphism

Rules for Function Overloading

Functions can be overloaded if they differ in:

1. Number of parameters
2. Type of parameters
3. Order of parameters

❖ Functions CANNOT be overloaded by: Return type alone

Example 1: Overloading by Number of Parameters

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

class Demo {
public:
    void add(int a, int b) {
        cout << "Sum = " << a + b << endl;
    }

    void add(int a, int b, int c) {
        cout << "Sum = " << a + b + c << endl;
    }
};
```

```
int main() {
    Demo d;
    d.add(10, 20);
    d.add(10, 20, 30);
    return 0;
}
```

Example 2: Overloading by Data Type

```
class Demo {
public:
    void show(int x) {
        cout << "Integer: " << x << endl;
    }

    void show(float x) {
        cout << "Float: " << x << endl;
    }
};
```

Example 3: Overloading by Order of Parameters

```
class Demo {
public:
    void display(int a, float b) {
        cout << "int-float" << endl;
    }

    void display(float a, int b) {
        cout << "float-int" << endl;
    }
};
```

Why Return Type Alone is Not Enough?

```
int add(int a, int b);
float add(int a, int b);  // ❌ ERROR
```

- ❖ Compiler cannot decide which function to call.

Function Overloading vs Virtual Function

Feature	Function Overloading	Virtual Function
Polymorphism type	Compile-time	Run-time
Binding	Static	Dynamic
Uses virtual	No	Yes
Inheritance required	No	Yes

******

3.10 Virtual functions Refer Materials 3.7 (Page Number 101)

******

3.11 Dynamic Binding through virtual functions

Dynamic binding means: The function call is resolved at run time based on the actual object type. Dynamic binding in C++ is achieved using virtual functions.

Role of Virtual Functions in Dynamic Binding

- Virtual function tells the compiler:

  "Decide the function call at runtime"
- Function call depends on **object**, not pointer
- Enables **runtime polymorphism**

Basic Concept

Base pointer

↓

Derived object

↓

Virtual function call

↓

Derived class function executed

Program Demonstrating Dynamic Binding

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual void display() {
        cout << "Display of Base class" << endl;
    }
};
class Derived : public Base {
public:
    void display() {
        cout << "Display of Derived class" << endl;
    }
};
int main() {
    Base* b;
    Derived d;

    b = &d;
    b->display();   // Dynamic binding

    return 0;
}
```

Output

```
Display of Derived class
```

Why Dynamic Binding is Needed?

Without dynamic binding:

- Base class function is always called
- Polymorphism cannot be achieved

With dynamic binding:

- Correct derived class function is called
- Behavior changes at runtime

Conditions for Dynamic Binding

Dynamic binding requires:

1. Inheritance
2. Virtual function
3. Base class pointer
4. Derived class object

******

3.12 Virtual function call mechanism

The **virtual function call mechanism** is the process by which: The correct virtual function is selected **at runtime** using a table of function addresses.

This mechanism uses:

- Virtual Table (v-table)
- Virtual Pointer (vptr)

Virtual Table (v-table)

A v-table is:

- A table created by the compiler
- Contains addresses of virtual functions
- Created once per class
- Used to support dynamic binding

Virtual Pointer (vptr)

A **vptr** is:

- A hidden pointer added to each object

- Points to the **v-table of its class**
- Automatically handled by compiler

❖ Programmers **do not write vptr explicitly**.

Basic Idea

Object created

↓

vptr is initialized

↓

vptr points to class v-table

↓

Function address is fetched

↓

Correct virtual function executed

Example Program

```
class Base {
public:
    virtual void show() { }
};

class Derived : public Base {
public:
    void show() { }
};
```

What Happens Internally? (Step-by-Step)

Assume

```
Base* b;
Derived d;
b = &d;
```

```
b->show();
```

Execution Steps

1. b points to a **Derived object**
2. Object contains a **vptr**
3. vptr points to **Derived's v-table**
4. v-table contains address of Derived::show()
5. That function is executed

❖ This is **runtime decision making**

Why Virtual Function Call Mechanism is Needed?

- Supports runtime polymorphism
- Enables dynamic binding
- Allows same function call to behave differently
- Makes inheritance more powerful

******

3.13 Pure virtual function

A **pure virtual function** is:

- A virtual function
- Has **no function body**
- Declared by assigning = 0

Syntax

virtual return_type function_name() = 0;

❖ It specifies **what to do**, not **how to do**.

Why Pure Virtual Functions Are Needed?

- To define a common interface
- To force derived classes to provide implementation
- To support complete runtime polymorphism
- To design frameworks and base models

Example

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

Explanation

- draw() has no implementation
- Any derived class **must override draw()**

******

3.14 Abstract Classes

An **abstract class** is:

- A class that contains **at least one pure virtual function**
- Cannot be used to create objects
- Used only as a **base class**

Abstract classes define **structure**, not objects.

Key Rules of Abstract Classes

- Object creation **not allowed**
- Can contain:
    - Data members
    - Normal member functions
    - Virtual functions
- Derived class must override all pure virtual functions

Example Program

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

class Shape {
public:
```

```
    virtual void draw() = 0;   // pure virtual function
};

class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing Circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() {
        cout << "Drawing Rectangle" << endl;
    }
};

int main() {
    Shape* s;

    Circle c;
    Rectangle r;

    s = &c;
    s->draw();

    s = &r;
    s->draw();
```

```
    return 0;
}
```

Output

```
Drawing Circle
Drawing Rectangle
```

Virtual Function vs Pure Virtual Function

Feature	Virtual Function	Pure Virtual Function
Has body	Yes	No
Syntax	`virtual f()`	`virtual f() = 0`
Class instantiation	Allowed	Not allowed
Mandatory override	No	Yes

Concrete Class vs Abstract Class

Feature	Concrete Class	Abstract Class
Object creation	Allowed	Not allowed
Pure virtual function	No	Yes
Used for	Objects	Inheritance

******

List of Practice Programs (Total: 24)

Inheritance & Class Hierarchy

1. Write a C++ program to demonstrate **single inheritance** using a base class Person and a derived class Student.
2. Write a C++ program to demonstrate **multilevel inheritance** with three classes in a hierarchy.
3. Write a C++ program to demonstrate **hierarchical inheritance** using one base class and two derived classes.

Different Forms of Inheritance

1. Write a C++ program to demonstrate **multiple inheritance** using two base classes and one derived class.
2. Write a C++ program to demonstrate **hybrid inheritance** (conceptual program).

Base and Derived Classes

1. Write a C++ program to define **base and derived classes** and access base class members using a derived class object.

Access to Base Class Members

1. Write a C++ program to demonstrate **public inheritance** and access base class members.
2. Write a C++ program to demonstrate **protected inheritance**.
3. Write a C++ program to demonstrate **private inheritance**.

Base and Derived Class Constructors

1. Write a C++ program to show the **order of execution of base and derived class constructors**.
2. Write a C++ program to demonstrate **parameterized base class constructor** using initializer list in the derived class.

Virtual Base Class

1. Write a C++ program to illustrate the **diamond problem** in multiple inheritance.
2. Write a C++ program to **solve the diamond problem using a virtual base class**.

Virtual Functions and Polymorphism

1. Write a C++ program to demonstrate **polymorphism without using virtual functions**.
2. Write a C++ program to demonstrate **polymorphism using virtual functions**.

Static and Dynamic Binding

1. Write a C++ program to demonstrate **static binding**.
2. Write a C++ program to demonstrate **dynamic binding using virtual functions**.

Function Overloading

1. Write a C++ program to demonstrate **function overloading** using different number of parameters.
2. Write a C++ program to demonstrate **function overloading using different data types**.

Dynamic Binding through Virtual Functions

1. Write a C++ program to demonstrate **dynamic binding through base class pointer and derived class object**.

Virtual Function Call Mechanism

1. Write a C++ program to explain **virtual function call mechanism** (conceptual demonstration using base pointer).

Pure Virtual Functions & Abstract Classes

1. Write a C++ program to demonstrate a **pure virtual function**.
2. Write a C++ program to demonstrate an **abstract class** with two derived classes.
3. Write a C++ program to achieve **runtime polymorphism using abstract class**.

****

UNIT III - QUESTION BANK

Part A (2 Marks Questions)

1. **Define inheritance (K1)**

   Inheritance is a mechanism in C++ by which a derived class acquires the properties and member functions of a base class.

2. **What is a class hierarchy? (K1)**

   A class hierarchy is an arrangement of classes where a base class is at the top and one or more derived classes are connected below it.

3. **State the need for class hierarchy. (K1)**

   Class hierarchy helps to represent real-world structure, reuse common data and functions, reduce duplication, and improve program design.

4. **Identify the IS-A relationship with an example. (K2)**

   The IS-A relationship indicates that a derived class is a type of base class.
   Example: Student is a Person.

5. **List the different forms of inheritance. (K1)**

   - Single inheritance
   - Multilevel inheritance
   - Multiple inheritance
   - Hierarchical inheritance and
   - Hybrid inheritance

6. **Explain single inheritance. (K2)**

   Single inheritance involves one base class and one derived class, where the derived class inherits from only one base class.

7. **Differentiate between multilevel and multiple inheritance. (K2)**

   In multilevel inheritance, a derived class becomes the base class for another class, whereas in multiple inheritance, a class inherits from more than one base class.

8. **Outline hierarchical inheritance. (K2)**

   Hierarchical inheritance involves one base class and multiple derived classes inheriting from the same base class.

9. **Define a base class. (K1)**

   A base class is a class that contains common data members and member functions which can be inherited by other classes.

10. **Define a derived class. (K1)**

    A derived class is a class that inherits the properties and member functions of a base class and may add new members.

11. **State the relationship between base and derived classes. (K1)**

    A derived class object contains both base class members and derived class members.

12. **List the access specifiers. (K1)**

    Public, protected, and private.

13. **Discuss access of base class members in public inheritance. (K2)**

    In public inheritance, public members of the base class remain public and protected members remain protected in the derived class.

14. **Why private members of a base class are not accessible in derived class. (K1)**

    Private members are accessible only inside the base class and cannot be accessed directly by derived classes.

15. **Define a Constructor. (K1)**

    A constructor is a special member function of a class that is automatically invoked when an object is created to initialize data members.

16. **State the order of constructor execution in inheritance. (K1)**

    When a derived class object is created, the base class constructor is executed first, followed by the derived class constructor.

17. **Why the base class constructor is called first. (K1)**

    The base class constructor is called first to initialize the base class part of the derived class object.

18. **What is a virtual base class?? (K1)**

    A virtual base class is a base class that ensures only one copy of the base class exists in multiple inheritance.

19. Identify the problem solved by a virtual base class. (K2)

A virtual base class solves the diamond problem caused by multiple inheritance.

20. Define a Polymorphism. (K1)

Polymorphism is the ability of a function to take different forms depending on the object type.

21. Define a Virtual Function. (K1)

A virtual function is a member function declared using the virtual keyword and resolved at runtime.

22. Infer the role of virtual functions. (K2)

Virtual functions support runtime polymorphism by ensuring that the correct derived class function is called based on the object type.

23. Define static binding. (K1)

Static binding is the process of resolving a function call at compile time.

24. Define dynamic binding. (K1)

Dynamic binding is the process of resolving a function call at runtime based on the actual object type.

25. Define an abstract class. (K1)

An abstract class is a class that contains at least one pure virtual function and cannot be used to create objects.

Part B (16 Marks Questions)

1. **Explain** inheritance and the concept of class hierarchy in C++. (K2)
2. **Describe** the different forms of inheritance supported by C++ with suitable examples. (K2)
3. **Discuss** the role of base and derived classes in inheritance. (K2)
4. **Differentiate** the access of base class members under public, protected, and private inheritance. (K2)
5. **Illustrate** the need for virtual functions in achieving polymorphism. (K2)
6. **Construct** a C++ program to demonstrate single and multilevel inheritance. (K3)

7. **Develop** a C++ program to show the order of execution of base and derived class constructors. (K3)
8. **Experiment** with a C++ program to achieve dynamic binding through virtual functions. (K3)
9. **Investigate** the use of a virtual base class to solve the diamond problem in multiple inheritance using C++. (K3)
10. **Apply** pure virtual functions to design an abstract class in C++. (K3)

Part C (16 Marks Questions)

1. **Develop** a C++ program for a vehicle management system where different vehicle types inherit common features and override behavior using virtual functions. (K3)
2. **Construct** a C++ program for an academic management system where Student and Teacher classes inherit common properties from a Person class. (K3)
3. **Use** inheritance and polymorphism to design a C++ program for a library system where different types of books share common features but differ in behavior. (K3)
4. **Organize** a simple C++ program for a shape system where Circle and Rectangle classes calculate area using an abstract base class. (K3)

Course In-charge	Module Coordinator	HoD/IT
Mrs. P. Saraswathi, AP-II	Mrs. M. Prabha, AP-II	Dr. R. Kavitha