What is Polymorphism

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects to be treated as instances of their parent class rather than their actual class. It enables a single interface to represent different underlying forms (data types). Polymorphism is of two types:

- **Compile-Time Polymorphism**: Also known as static polymorphism, it is achieved through function overloading and operator overloading. The function or operator to be executed is determined at compile time.
- **Run-Time Polymorphism**: Also known as dynamic polymorphism, it is achieved through function overriding and virtual functions. The function to be executed is determined at runtime.

Compile-Time Polymorphism

Compile-time polymorphism is resolved during the compilation process. It includes:

- **Constructor Overloading**: Defining multiple constructors with different parameter lists.
- **Function Overloading**: Defining multiple functions with the same name but different parameters.
- **Operator Overloading**: Defining custom behavior for operators when used with user-defined types (classes or structures).

1. Constructor Overloading

Constructor overloading allows a class to have multiple constructors with different parameter lists. The appropriate constructor is selected based on the arguments provided during object creation.

Example:

```
class Example {
private:
    int x;
    double y;

public:
    Example() { // Default constructor
        x = 0;
        y = 0.0;
    }

    Example(int a) { // Constructor with one parameter
        x = a;
        y = 0.0;
    }
```

```
    Example(int a, double b) { // Constructor with two parameters
        x = a;
        y = b;
    }
};
```

2. Function Overloading

Function overloading allows multiple functions to have the same name but differ in the number or type of parameters. The compiler decides which function to call based on the arguments passed.

Example:

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

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Overloaded function to add three integers
int add(int a, int b, int c) {
    return a + b + c;
}

// Overloaded function to add two doubles
double add(double a, double b) {
    return a + b;
}

int main() {
    cout << "Sum of 2 and 3: " << add(2, 3) << endl;
    cout << "Sum of 2, 3, and 4: " << add(2, 3, 4) << endl;
    cout << "Sum of 2.5 and 3.5: " << add(2.5, 3.5) << endl;
    return 0;
}
```

3. Operator Overloading

- An operator is said to be overloaded if it is defined for multiple types. In other words, overloading an operator means making the operator significant for a new type.
- Operators can be used with user-defined types as well. Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects. Example: The effect of **+** operator can be stipulated for the objects of a particular class.
- Operator Function Syntax: To overload an operator, an appropriate operator function is required.

```
returntype operator op (arg_list) {
    // Function body
}
```

  - `returntype`: The type of value returned by the operation.
  - `op`: The operator being overloaded (e.g., `+`, `-`, «, »).
  - `op` is preceded by the keyword `operator`.

Operator Overloading as Member Functions

If the operator function is defined as a method inside the class, the left operand must always be an object of the class. The operator function is called for this object. The right operand is passed as an argument to the method.

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

class Point {
private:
    int x, y;
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}

    // Overloading the + operator
    Point operator+(const Point& obj) const {
        Point temp;
        temp.x = x + obj.x;
        temp.y = y + obj.y;
        return temp;
    }
    void display() const {
        cout << "(" << x << ", " << y << ")" << endl;
    }
};
```

```
int main() {
    Point point1(10, 20), point2(30, 40);
    Point point3 = point1 + point2; // Using overloaded + operator
    point3.display();
    return 0;
}
```

Overloading Insertion («) and Extraction (») Operators

Insertion and extraction operators must be overloaded as non-member functions (global or friend functions). If the data inside the class is private, the overloaded functions must be declared as `friend` functions.

```
class Point {
private:
    int x, y;
    friend ostream& operator<<(ostream& os, const Point& point1);
    friend istream& operator>>(istream& in, Point& point1);
public:
    Point(int a = 0, int b = 0) : x(a), y(b) {}
};

// Overloading << operator
ostream& operator<<(ostream& os, const Point& point1) {
    os << "(" << point1.x << ", " << point1.y << ")" << endl;
    return os;
}

// Overloading >> operator
istream& operator>>(istream& in, Point& point1) {
    cout << "Enter value of x co-ordinate: ";
    in >> point1.x;
    cout << "Enter value of y co-ordinate: ";
    in >> point1.y;
    return in;
}

int main() {
    Point point1;
    cin >> point1;
    cout << point1;
    return 0;
}
```

Separating Interface from Implementation (Writing Code in Multiple Files)

In large projects, it is a good practice to separate the **interface** (declaration) from the **implementation** (definition). This improves code readability, reusability, and maintainability. In C++, this is achieved by:

- Placing class declarations in a header file (`.h`).
- Defining member functions in a source file (`.cpp`).

Example

Step 1: Create a header file (`math_operations.h`)

```
// math_operations.h
#ifndef MATH_OPERATIONS_H
#define MATH_OPERATIONS_H

class MathOperations {
public:
    int add(int a, int b);
    double add(double a, double b);
};

#endif
```

Step 2: Create a source file (`math_operations.cpp`)

```
// math_operations.cpp
#include "math_operations.h"

int MathOperations::add(int a, int b) {
    return a + b;
}

double MathOperations::add(double a, double b) {
    return a + b;
}
```

Step 3: Create the main file (`main.cpp`)

```cpp
// main.cpp
#include <iostream>
#include "math_operations.h"
using namespace std;

int main() {
    MathOperations math;
    cout << "Sum of 2 and 3: " << math.add(2, 3) << endl;
    cout << "Sum of 2.5 and 3.5: " << math.add(2.5, 3.5) << endl;
    return 0;
}
```

Inheritance in C++

Inheritance is a mechanism in C++ that allows a class to inherit properties and behaviors (methods) from another class. The class that inherits is called the **derived class**, and the class being inherited from is called the **base class**.

Modes of Inheritance in C++

Mode of inheritance controls the access level of the inherited members of the base class in the derived class. In C++, there are three modes of inheritance:

1. Public Inheritance

Public inheritance is a fundamental concept in object-oriented programming (OOP) where a derived class inherits attributes and behaviors (methods) from another parent class.
The access levels of the base class members are preserved in the derived class.

- Public members of the base class remain public in the derived class.
- Protected members of the base class remain protected in the derived class.
- Private members of the base class are not accessible in the derived class.

Example:

```cpp
#include <iostream>

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

class Derived : public Base {
public:
    void display() {
        std::cout << "Derived class method" << std::endl;
    }
};

int main() {
    Derived obj;
    obj.show();   // Accessible (inherited as public)
    obj.display();
    return 0;
}
```

2. Protected Inheritance

Protected inheritance is a form of inheritance in which a derived class inherits from a base class using the ‘protected‘ access specifier. This means that the base class’s public and protected members become protected in the derived class.
The access levels of the base class members in the derived class are modified as follows:

- Public members of the base class remain protected in the derived class.
- Protected members of the base class remain protected in the derived class.
- Private members of the base class are not accessible in the derived class.

Example:

```cpp
#include <iostream>

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

class Derived : protected Base {
public:
    void display() {
        std::cout << "Derived class method" << std::endl;
        show(); // Allowed since it's now protected
    }
};

int main() {
    Derived obj;
    // obj.show(); // Error: show() is protected in Derived
    obj.display();
    return 0;
}
```

3. Private Inheritance

Private inheritance is a type of inheritance where a derived class inherits from a base class using the 'private' access specifier. This means that all accessible members of the base class become private in the derived class. The access levels of the base class members in the derived class are modified as follows:

- Public members of the base class remain private in the derived class.
- Protected members of the base class remain private in the derived class.
- Private members of the base class are not accessible in the derived class.

Example:

```cpp
#include <iostream>

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

class Derived : private Base {
public:
    void display() {
        std::cout << "Derived class method" << std::endl;
        show(); // Allowed since it's now private
    }
};

int main() {
    Derived obj;
    // obj.show(); // Error: show() is private in Derived
    obj.display();
    return 0;
}
```

Types of Inheritance in C++

The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have five types of inheritances:

1. Single Inheritance

In single inheritance, a derived class inherits from a single base class. This is the simplest form of inheritance.

Example:

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

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

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

int main() {
    Derived obj;
    obj.show();    // Inherited from Base
    obj.display(); // Defined in Derived
    return 0;
}
```

2. Multiple Inheritance

In multiple inheritance, a derived class inherits from more than one base class. **Example:**

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

class Base1 {
public:
    void show1() {
        cout << "Base1 class method" << endl;
    }
};

class Base2 {
public:
    void show2() {
        cout << "Base2 class method" << endl;
    }
};

class Derived : public Base1, public Base2 {
public:
    void display() {
        cout << "Derived class method" << endl;
    }
};
```

```
int main() {
    Derived obj;
    obj.show1();   // Inherited from Base1
    obj.show2();   // Inherited from Base2
    obj.display(); // Defined in Derived
    return 0;
}
```

3. Multilevel Inheritance

In multilevel inheritance, a derived class acts as a base class for another derived class, forming a chain of inheritance.

Example:

```
class Grandparent {
public:
    void show() {
        cout << "Grandparent class method" << endl;
    }
};

class Parent : public Grandparent {
public:
    void display() {
        cout << "Parent class method" << endl;
    }
};

class Child : public Parent {
public:
    void print() {
        cout << "Child class method" << endl;
    }
};

int main() {
    Child obj;
    obj.show();    // Inherited from Grandparent
    obj.display(); // Inherited from Parent
    obj.print();   // Defined in Child
}
```

4. Hierarchical Inheritance

In hierarchical inheritance, multiple classes inherit from a single base class.

Example:

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

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

class Derived1 : public Base {
public:
    void display1() {
        cout << "Derived1 class method" << endl;
    }
};

class Derived2 : public Base {
public:
    void display2() {
        cout << "Derived2 class method" << endl;
    }
};

int main() {
    Derived1 obj1;
    obj1.show();     // Inherited from Base
    obj1.display1(); // Defined in Derived1

    Derived2 obj2;
    obj2.show();     // Inherited from Base
    obj2.display2(); // Defined in Derived2
    return 0;
}
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. It can involve multiple, hierarchical, or multilevel inheritance together in a single program.

Example:

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

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

class Derived1 : public Base {
public:
    void display1() {
        cout << "Derived1 class method" << endl;
    }
};

class Derived2 : public Base {
public:
    void display2() {
        cout << "Derived2 class method" << endl;
    }
};

class Derived3 : public Derived1, public Derived2 {
public:
    void display3() {
        cout << "Derived3 class method" << endl;
    }
};
int main() {
    Derived3 obj;
    // obj.show();  // Causes ambiguity due to multiple inheritance from Base
    obj.display1();
    obj.display2();
    obj.display3();
    return 0; }
```

Lab exercises

Exercise 1

Geometry Toolkit: You are tasked with developing a program for a **geometry toolkit** that computes the perimeter of different geometric shapes using function overloading. Implement an overloaded function `calculatePerimeter()` that works as follows:

- **For a Circle** – Accepts a single parameter (radius) and returns the perimeter (circumference).
- **For a Rectangle** – Accepts two parameters (length and width) and returns the perimeter.
- **For a Triangle** – Accepts three parameters (lengths of all three sides) and returns the perimeter.
- **For a Polygon** – Accepts an array of n side lengths and returns the perimeter.

Hint: Perimeter Formulas

Circle: $P = 2\pi r$

Rectangle: $P = 2 \times (\text{length} + \text{width})$

Triangle: $P = a + b + c$

Polygon: $P = \sum (\text{all side lengths})$

Your program should prompt the user to choose a shape, input the required values, and display the corresponding perimeter.

Exercise 2

Complex Numbers. Define a class for complex numbers. A complex number is a number of the form:

$$a + b \cdot i$$

where a and b are numbers of type `double`, and i represents the quantity $\sqrt{-1}$. Represent a complex number as two values of type `double`. Name the member variables `real` and `imaginary`. The variable representing the coefficient of i should be called `imaginary`. Call the class `Complex`.

Include the following constructors:

- A constructor with two parameters of type `double` that initializes the member variables to given values.
- A constructor with a single parameter of type `double` (named `realPart`) that initializes the object to `realPart` $+0i$.
- A default constructor that initializes an object to $0 + 0i$.

Overload the following operators so they correctly apply to the `Complex` class:

- `+` (addition)
- `-` (subtraction)
- `*` (multiplication)
- `==` (equality comparison)

- » (input stream)
- « (output stream)

Write a test program to validate your class.

Hint: To add or subtract two complex numbers, simply add or subtract their respective real and imaginary parts.

The product of two complex numbers follows the formula:

$$(a + b \cdot i) \times (c + d \cdot i) = (a \cdot c - b \cdot d) + (a \cdot d + b \cdot c) \cdot i$$

Exercise 3

Bank Account Management System: You are tasked with developing a **Bank Account Management System** using the principle of separating interface from implementation in C++. The system should allow users to create a bank account, deposit money, withdraw money, and view transaction logs. To achieve this, design a class named `BankAccount` that provides essential functionalities. Implement the class by separating the interface from the implementation.

Class Attributes:

- `accountHolder` (string): Stores the name of the account holder.
- `accountNumber` (int): Unique account number.
- `balance` (double): Stores the current balance.
- `transactionLog` (vector<string>): Stores a record of transactions.

Member Functions:

- `BankAccount(std::string name, int accNumber, double balance)`: Initializes the account with the **holder's name, account number, and an optional initial balance** (default: 0.0).
- `void deposit(double amount)`: Adds money to the account and logs the transaction.
- `bool withdraw(double amount)`: Deducts money if the balance is sufficient and logs the transaction. Displays an error for insufficient funds.
- `void displayAccountDetails() const`: Displays account holder details and current balance.
- `void showTransactionLog() const`: Displays the transaction history of the account.

The `main.cpp` file should be able to perform the following functions:

- Create an account for a user.
- Perform a few transactions (deposits and withdrawals).
- Display the account details and transaction log.

Constraints:

- The withdrawal amount should not exceed the current balance.
- Depositing or withdrawing a negative amount should not be allowed.
- Use separate files (`BankAccount.h`, `BankAccount.cpp`, and `main.cpp`) to follow the separation of interface from implementation principle.

Exercise 4

Library Management System: Develop a **Library Management System** that should manage different types of publications, allowing users to store and retrieve details about books, research papers, and digital resources.

(a) Define Classes Using Inheritance

Create a **base class `Publication`** and derive the following three classes from it:

- **`Book`**: Represents physical or digital books.
- **`ResearchPaper`**: Represents academic research papers.
- **`DigitalResource`**: Represents online articles, e-books, or videos.

(b) Implement the Base Class (`Publication`)

The `Publication` class should have:

- **Attributes:** `title` (string), `author` (string), and `yearPublished` (int).
- A constructor to initialize all attributes.
- A **`printDetails()`** method that prints all publication details.

(c) Implement the Derived Classes

Each derived class must:

- Have **2-3 additional attributes**:
  - **`Book`**: `ISBN` (string), `totalPages` (int).
  - **`ResearchPaper`**: `journalName` (string), `impactFactor` (double).
  - **`DigitalResource`**: `URL` (string), `fileSizeMB` (double).
- A constructor that **initializes both inherited and new attributes**
- A **`printDetails()`** method that prints all attributes, including inherited ones.

(d) Implement the `main()` Function

- Create **at least 5 instances** of each derived class (`Book`, `ResearchPaper`, `DigitalResource`).
- Store them in an **array of pointers** to the base class (`Publication*`).
- Iterate through the array and call the **`printDetails()`** method for each object.