WENDY WAHOME CAT PRESENTATION:

PART A

1. Using a well labeled diagram, explain the steps of creating a system using OOP principles

1. Identify Objects:

The first step is to identify the main objects in the system. An object represents an entity in the problem domain. For example, in an e-commerce system, objects might include customers, orders, and products.

2. Define Classes:

For each identified object, create a corresponding class. A class is a blueprint or template that defines the properties and behaviors of objects. For example, a customer class might define attributes such as name, email, and address, as well as methods for placing orders and updating account information.

3. Identify Attributes:

Determine the attributes (characteristics or properties) of each class. These attributes define the state of the objects. For example, a product class might have attributes such as price, description, and quantity.

4. Define Methods:

Identify the methods (functions or procedures) that operate on the data and behaviors of each class. Methods define the actions or operations that can be performed on objects of a class. For example, a customer class might have methods for adding items to a shopping cart, checking out, and viewing order history.

5. Encapsulation:

Encapsulate the attributes and methods within the classes. Encapsulation restricts access to certain components of an object and protects the integrity of the object's state. This helps to prevent unintended changes to the object's data and behavior.

6. Define Relationships:

Identify relationships between classes. These relationships can include associations (e.g., one-to-one, one-to-many), inheritance (e.g., superclass and subclass relationships), and composition (e.g., one class contains objects of another class). For example, a customer might have a one-to-many relationship with orders, and an order might contain one or more products.

7. Create Objects:

Instantiate objects from the defined classes. Objects are instances of classes and represent specific entities in the system. For example, a customer object might represent a particular customer, with its own unique data and behavior.

8. Inheritance:

Apply inheritance where appropriate. Inheritance allows a class to inherit properties and behaviors from another class, promoting code reuse and establishing an "is-a" relationship. For example, a product might inherit from a base class that defines common attributes and methods for all products.

9. Polymorphism:

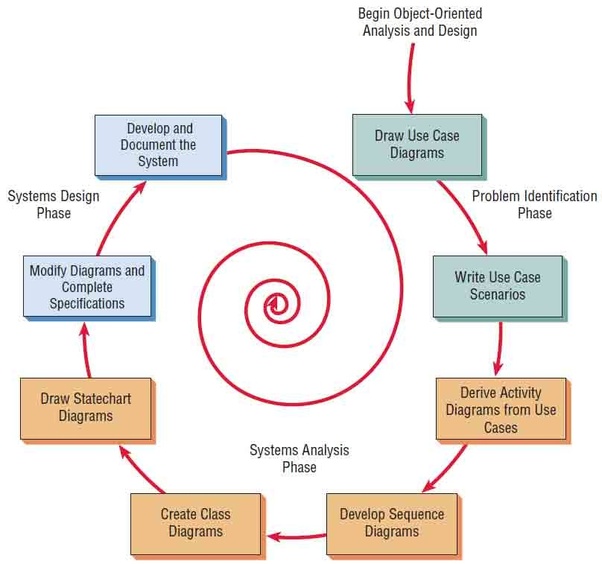
Implement polymorphism, which allows objects of different classes to be treated as objects of a common superclass. Polymorphism enables flexibility and extensibility in the system. For example, a discount might be applied to both products and orders, even though they are different classes, by treating them as objects of a common superclass.

10. Abstraction:

Use abstraction to hide unnecessary details and focus on essential features. Abstract classes and interfaces can be employed to define a common set of methods that subclasses must implement. This helps to simplify the code and make it easier to understand and modify.

11. Testing:

Test the system to ensure that it behaves as expected. Unit testing, integration testing, and system testing are essential steps in verifying the correctness and reliability of the implemented system. This helps to catch bugs and ensure that the system meets all requirements and specifications.



ii. What is the Object Modeling Techniques (OMT).

Object Modeling Techniques (OMT) is a methodology used for designing and modeling software systems based on the principles of object-oriented programming. It provides a comprehensive approach to software development that involves identifying, analyzing, and representing the objects, attributes, and relationships involved in a system.

iii. Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP).

In summary, OOAD is a more comprehensive approach to software engineering that involves a number of There are two software engineering approaches for designing and implementing software systems, namely Object-oriented analysis and design (OOAD) and object analysis and design (OOP). OOAD involves applying object-oriented concepts and techniques such as design patterns, UML diagrams, and use cases to design and implement software systems.

In contrast, OOP is focused on converting requirements into an implementation specification by identifying objects, defining their behaviors, and specifying which objects can activate specific behaviors on other objects. Overall, OOAD is a more comprehensive approach to software engineering, while OOP is more focused on the implementation specification. techniques and practices, while OOP is a more focused approach that is primarily concerned with converting requirements into an implementation specification ¹³.

iv. Discuss Mian goals of UML.

Object-oriented analysis and design (OOAD) and object analysis and design (OOP) are two software engineering methodologies that are used to design and implement software systems. OOAD is a software engineering methodology that involves using object-oriented concepts to design and implement software systems. OOAD involves a number of techniques and practices, including object-oriented programming, design patterns, UML diagrams, and use cases ¹.

On the other hand, Object analysis and design (OOP) is a process of converting requirements into an implementation specification. The designer must name the objects, define the behaviors, and formally specify which objects can activate specific behaviors on other objects .

v. DESCRIBE three advantages of using object oriented to develop an information system.

1. . Developing an information system using object-oriented programming involves creating classes that encapsulate both data and functions. This approach enables modularity, making troubleshooting more efficient by isolating problem areas more easily. Furthermore, it promotes teamwork by allowing multiple team members to work concurrently without duplicating functionality, as each object functions independently.
2. Inheritance is a feature of object-oriented programming that allows for the reuse of code. Rather than building new objects from scratch, a generic class can be defined, and subclasses can be created to inherit the characteristics of the generic class while still maintaining their unique attributes and functions. This technique saves time and encourages code reuse.
3. Object-oriented programming emphasizes reusability by permitting the reuse of existing facilities (classes) rather than the creation of new ones. By creating self-contained, modular objects, they can be effortlessly integrated into larger systems. Additionally, object-oriented programming systems are scalable from small to large projects, making maintenance more manageable.

vi. Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept.

Here are the explanations of the terms you asked for:

a) A constructor is a method that is specially designed to initialize objects. It gets invoked when a new object of a class is created. The constructor has an identical name as that of the class and doesn't have any return type. Java allows constructors to be overloaded, which implies that a class can have multiple constructors with distinct parameters. Here is an example of a constructor in Java:

```java

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

}

```

b. Object: An object is an instance of a class. It is created using the `new` keyword followed by the name of the class. Here is an example of creating an object in Java:

```java

Person person = new Person("John", 30);

```

c. Destructor: Unlike C++, Java does not have destructors. Instead, Java has a garbage collector that automatically frees up memory when an object is no longer being used.

d. Polymorphism: Polymorphism is the ability of an object to take on many forms. In Java, polymorphism is achieved through method overriding and method overloading. Method overriding is when a subclass provides a specific implementation of a method that is already provided by its parent class. Method overloading is when a class has two or more methods with the same name but different parameters. Here is an example of polymorphism in Java:

```java

public class Animal {

public void makeSound() {

System.out.println("The animal makes a sound");

}

}

public class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("The dog barks");

}

}

public class Cat extends Animal {

@Override

public void makeSound() {

System.out.println("The cat meows");

}

}

public class Main {

public static void main(String[] args) {

Animal animal1 = new Dog();

Animal animal2 = new Cat();

animal1.makeSound();

animal2.makeSound();

}

}

```

e. Class: A class is a blueprint for creating objects. It defines a set of attributes and methods that are common to all objects of that class. Here is an example of a class in Java:

```java

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public String getName() {

return name;

}

public int getAge() {

return age;

}

}

```

f. Inheritance: Inheritance is a mechanism in which one class acquires the properties and methods of another class. The class that is being inherited from is called the parent class or superclass, and the class that is inheriting is called the child class or subclass. In Java, inheritance is achieved using the `extends` keyword. Here is an example of inheritance in Java:

```java

public class Animal {

public void eat() {

System.out.println("The animal eats");

}

}

public class Dog extends Animal {

public void bark() {

System.out.println("The dog barks");

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat();

dog.bark();

}

}

VI): EXPLAIN the three types of associations (relationships) between objects in object-oriented.

Association:

Association represents a bi-directional relationship between two or more classes. It indicates that objects of one class are related to objects of another class.

Example: In a school system, there can be an association between the "Teacher" class and the "Student" class, indicating that teachers interact with students. This association is often represented by a line connecting the participating classes, with optional multiplicity annotations to specify the number of instances involved.

java

public class Teacher {

// Class definition

}

public class Student {

// Class definition

}

// Association between Teacher and Student

Aggregation:

Aggregation is a specialized form of association where one class represents a whole and another class represents a part. It denotes a "whole-part" relationship, where the part can exist independently of the whole.

Example: In a car manufacturing system, a "Car" class can be aggregated with a "Wheel" class. The wheels can exist independently (e.g., in a warehouse), but they are part of a car when aggregated.

java

public class Car {

List<Wheel> wheels; // Aggregation

}

public class Wheel {

// Class definition

}

Composition:

Composition is a stronger form of aggregation, where the part is considered an integral part of the whole. If the whole is destroyed, all its parts are also destroyed.

Example: In a computer system, a "Computer" class can be composed of a "CPU" class, "Memory" class, and "HardDrive" class. When the computer is destroyed, its components are also destroyed.

java

public class Computer {

CPU cpu; // Composition

Memory memory; // Composition

HardDrive hd; // Composition

}

public class CPU {

// Class definition

}

public class Memory {

// Class definition

}

public class HardDrive {

// Class definition

}

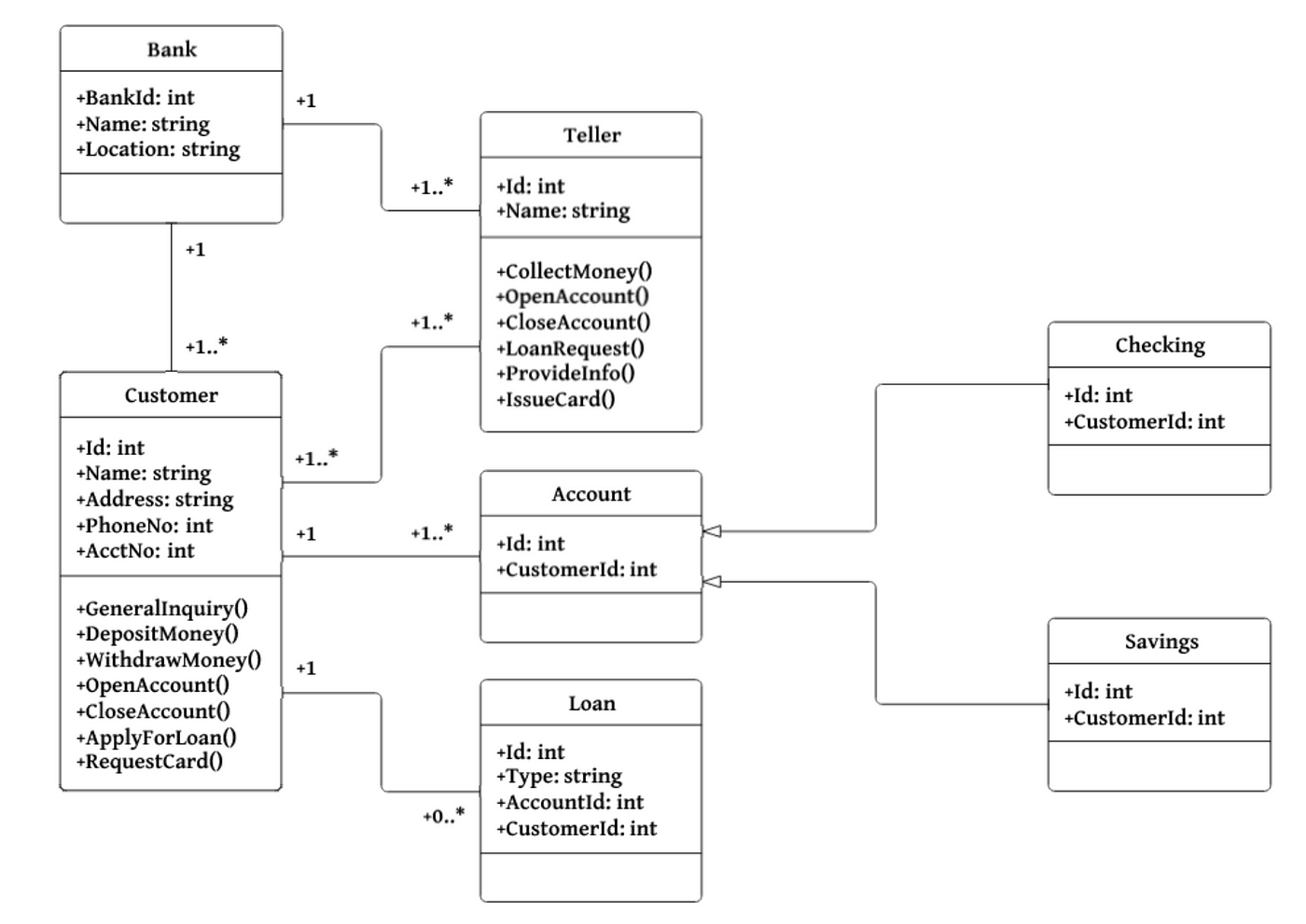
VII). What do you mean by class diagram? Where it is used and also discuss the steps to draw the class diagram with any one example.

The first step that needs to be taken is to recognize the classes that are a part of the system. A class serves as a blueprint for making objects that have shared traits and actions. For instance, if we are designing a system for a library, we may identify classes like `Book`, `Author`, `Library`, and `Member`.

2. The following step is to identify the relationships that exist between the identified classes. There are several types of relationships that can exist between classes, such as association, aggregation, and composition. For example, an `Author` can write many `Books`, and a `Book` can be written by an `Author`. This exemplifies an association relationship.

3. Once the classes and their relationships have been identified, the next step is to add techniques and attributes to the classes. Attributes are the characteristics of a class, while methods are the actions of a class. For example, a `Book` class might have attributes like `title`, `author`, and `ISBN`, and methods like `checkOut()` and `return()`.

4. The final step in this process is to illustrate the class diagram. Each class is represented as a rectangle with three compartments. The name of the class appears in the top compartment, the attributes of the class are in the middle compartment, and the methods of the class are in the bottom compartment. The relationships between the classes are represented by lines connecting the classes. For instance, the class diagram for a Bank system may look something like this:



vii. Given that you are creating area and perimeter calculator using C++, to computer area and perimeter of various shaped like Circles, Rectangle, Triangle and Square, use well written code to explain and implement the calculator using the following OOP concepts.

#include<iostream>

#include<cmath>

class Shape {

public:

virtual double calculateArea() const = 0;

virtual double calculatePerimeter() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

Circle(double r) : radius(r) {}

double calculateArea() const override {

return 3.14 \* radius \* radius;

}

double calculatePerimeter() const override {

return 2 \* 3.14 \* radius;

}

};

class Rectangle : public Shape {

private:

double length;

double width;

public:

Rectangle(double l, double w) : length(l), width(w) {}

double calculateArea() const override {

return length \* width;

}

double calculatePerimeter() const override {

return 2 \* (length + width);

}

};

class Triangle : public Shape {

private:

double side1;

double side2;

double side3;

public:

Triangle(double s1, double s2, double s3) : side1(s1), side2(s2), side3(s3) {}

double calculateArea() const override {

double s = (side1 + side2 + side3) / 2;

return sqrt(s \* (s - side1) \* (s - side2) \* (s - side3));

}

double calculatePerimeter() const override {

return side1 + side2 + side3;

}

};

class Square : public Rectangle {

public:

Square(double side) : Rectangle(side, side) {}

}

void displayResults(const Shape& shape) {

std::cout << "Area: " << shape.calculateArea() << std::endl;

std::cout << "Perimeter: " << shape.calculatePerimeter() << std::endl;

}

int main() {

Circle circle(5.0);

Rectangle rectangle(4.0, 6.0);

Triangle triangle(3.0, 4.0, 5.0);

Square square(4.0);

displayResults(circle);

displayResults(square);

displayResults(rectangle);

displayResults(triangle);

return 0;

}

Inheritance:

Single Inheritance: Circle, Rectangle, Triangle, and Square inherit from the abstract class Shape.

Multiple Inheritance: Square inherits from Rectangle, demonstrating multiple inheritance.

Hierarchical Inheritance: Rectangle and Triangle inherit from Shape, representing hierarchical inheritance.

Friend Functions:

The displayResults function is a friend function that can access the private members of the Shape class to display the calculated area and perimeter.

Method Overloading and Method Overriding:

Method Overloading: The calculateArea and calculatePerimeter methods are overloaded in each derived class to provide specific implementations for each shape.

Method Overriding: The override keyword is used to indicate that these methods in the derived classes override the virtual functions in the base class.

Late Binding and Early Binding:

Late Binding: Achieved through the use of virtual functions in the Shape class, allowing the appropriate function to be called based on the actual type of the object during runtime (polymorphism).

Early Binding: Occurs for non-virtual functions and functions called on objects with known types during compile-time.

Abstract Class and Pure Functions:

The Shape class is an abstract class containing pure virtual functions (calculateArea and calculatePerimeter). Objects of this class cannot be instantiated, but it serves as an interface for the derived classes.

Viii .Using a program written in C++, differentiate between the following.

a. Function overloading and operator overloading

b. Pass by value and pass by reference

c. Parameters and arguments

#include<iostream>

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

class Complex {

private:

double real;

double imag;

public:

Complex(double r, double i) : real(r), imag(i) {}

Complex operator+(const Complex& other) const {

return Complex(real + other.real, imag + other.imag);

}

void display() const {

std::cout << real << " + " << imag << "i" << std::endl;

}

};

void passByValue(int x) {

x = x \* 2;

}

void passByReference(int &x) {

x = x \* 2;

}

int main() {

std::cout << "Function Overloading:" << std::endl;

std::cout << "Sum (int): " << add(3, 5) << std::endl;

std::cout << "Sum (double): " << add(3.5, 5.5) << std::endl;

std::cout << "\nOperator Overloading:" << std::endl;

Complex a(1.0, 2.0);

Complex b(2.5, 3.5);

Complex result = a + b;

std::cout << "Result: ";

result.display();

std::cout << "\nPass by Value:" << std::endl;

int value = 10;

passByValue(value);

std::cout << "Value after passByValue: " << value << std::endl;

std::cout << "\nPass by Reference:" << std::endl;

int reference = 10;

passByReference(reference);

std::cout << "Value after passByReference: " << reference << std::endl;

return 0;

}

Function overloading and operator overloading are two concepts in programming. Function overloading refers to the use of multiple functions with the same name but different parameter types or the number of parameters. For example, you can have two different functions named add, where one takes in two integers as parameters, and the other takes in two doubles. Operator overloading, on the other hand, involves defining custom behavior for operators based on the context of their operands. For instance, you can overload the + operator for a Complex class by defining the operator+ function.

Pass by value and pass by reference are two ways of passing arguments to a function. When you pass an argument by value, you're passing the actual value of a variable to a function. Changes to the parameter inside the function do not affect the original value. For example, if you have a function called passByValue that takes an integer parameter, changes made to that parameter inside the function will not affect the original value of the variable that was passed to the function.

On the other hand, when you pass an argument by reference, you're passing the memory address (reference) of a variable to a function. Changes to the parameter inside the function will affect the original value. For instance, if you have a function called passByReference that takes an integer reference as a parameter, changes made to that parameter inside the function will affect the original value of the variable that was passed to the function.

Parameters and arguments are two related concepts in programming. Parameters are variables declared in a function's signature, and they act as placeholders for the values that will be passed to the function. For example, if you have a function called add that takes two integer parameters, a and b, a and b are the parameters. Arguments, on the other hand, are the actual values passed to a function when it is called. They match the types and order of the function's parameters. For example, if you call the add function with the arguments 3 and 5, 3 and 5 are the arguments.

VIII): Create a new class called CalculateG

#include<iostream>

class CalculateG {

public:

int main() {

double gravity = -9.81;

double fallingTime = 30.0;

double initialVelocity = 0.0;

double finalVelocity = 0.0;

double initialPosition = 0.0;

double finalPosition = 0.0;

finalPosition = 0.5 \* gravity \* fallingTime \* fallingTime + initialVelocity \* fallingTime + initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

std::cout << "The object's position after " << fallingTime << " seconds is " << finalPosition << " m." << std::endl;

std::cout << "The object's velocity after " << fallingTime << " seconds is " << finalVelocity << " m/s." << std::endl;

return 0;

}

};

int main() {

CalculateG calculator;

calculator.main();

return 0;

}

X): Extend datatype class with the following code:

public class CalculateG {

public double multi(double a, double b) {

return a \* b;

}

public double powerToSquare(double x) {

return Math.pow(x, 2);

}

public double summation(double a, double b) {

return a + b;

}

public void outline(double result) {

System.out.println("Result: " + result);

}

public static void main(String[] args) {

CalculateG calculator = new CalculateG();

double gravity = -9.81;

double fallingTime = 30.0;

double initialVelocity = 0.0;

double finalVelocity = 0.0;

double initialPosition = 0.0;

double finalPosition = 0.0;

finalPosition = 0.5 \* gravity \* calculator.powerToSquare(fallingTime) + initialVelocity \* fallingTime

+ initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

calculator.outline(finalPosition);

calculator.outline(finalVelocity);

}

}

XI): Create methods for multiplication, powering to square, summation and printing out a result in CalculateG class.

public class CalculateG {

public double multiply(double a, double b) {

return a \* b;

}

public double powerToSquare(double x) {

return Math.pow(x, 2);

}

public double summation(double a, double b) {

return a + b;

}

public void printResult(double result) {

System.out.println("Result: " + result);

}

public static void main(String[] args) {

CalculateG calculator = new CalculateG();

double gravity = -9.81;

double fallingTime = 30.0;

double initialVelocity = 0.0;

double finalVelocity = 0.0;

double initialPosition = 0.0;

double finalPosition = 0.0;

finalPosition = 0.5 \* gravity \* calculator.powerToSquare(fallingTime) + initialVelocity \* fallingTime

+ initialPosition;

finalVelocity = gravity \* fallingTime + initialVelocity;

calculator.printResult(finalPosition);

calculator.printResult(finalVelocity);

}

}

PART B

**QUESTIONN 1:**

#include<iostream>

long long findSumOfEvenFibonacciTerms(int limit) {

long long sum = 0;

long long firstTerm = 1, secondTerm = 2;

while (secondTerm <= limit) {

if (secondTerm % 2 == 0) {

sum += secondTerm;

}

long long nextTerm = firstTerm + secondTerm;

firstTerm = secondTerm;

secondTerm = nextTerm;

}

return sum;

}

int main() {

int limit = 4000000;

long long result = findSumOfEvenFibonacciTerms(limit);

std::cout << "The sum of even-valued Fibonacci terms below " << limit << " is: " << result << std::endl;

return 0;

}

**QUESTION THREE:**

#include<iostream>

int main() {

const int size = 15;

int originalArray[size];

std::cout << "Enter 15 integer values:\n";

for (int i = 0; i < size; ++i) {

std::cout << "Value " << (i + 1) << ": ";

std::cin >> originalArray[i];

}

std::cout << "\nValues stored in the array:\n";

for (int i = 0; i < size; ++i) {

std::cout << originalArray[i] << " ";

}

int searchNumber;

std::cout << "\nEnter a number to search in the array: ";

std::cin >> searchNumber;

bool numberFound = false;

int foundIndex = -1;

for (int i = 0; i < size; ++i) {

if (originalArray[i] == searchNumber) {

numberFound = true;

foundIndex = i;

break;

}

}

if (numberFound) {

std::cout << "The number found at index " << foundIndex << std::endl;

} else {

std::cout << "Number not found in this array.\n";

}

int reversedArray[size];

for (int i = 0; i < size; ++i) {

reversedArray[i] = originalArray[size - 1 - i];

}

std::cout << "\nElements of the new array in reverse order:\n";

for (int i = 0; i < size; ++i) {

std::cout << reversedArray[i] << " ";

}

int sum = 0;

long long product = 1;

for (int i = 0; i < size; ++i) {

sum += originalArray[i];

product \*= originalArray[i];

}

std::cout << "\n\nSum of elements: " << sum << std::endl;

std::cout << "Product of elements: " << product << std::endl;

return 0;

}