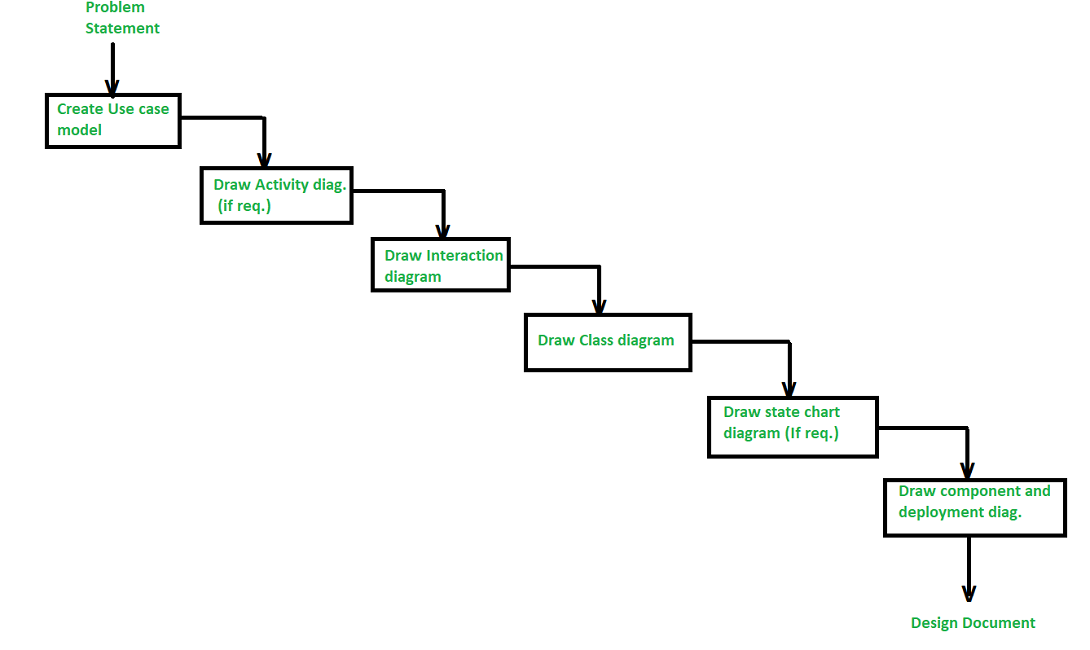
**Name: Beatrice Muthoni Kariuki**

**Reg No: Sct121-0943/2022**

**Q1: Using a well-labeled diagram, explain the steps of creating a system using OOP principles**



**Identify Objects:**

Begin by identifying the main objects in the system. Objects are instances of classes and represent entities in the problem domain.

**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.

**Identify Attributes:**

Determine the attributes (characteristics or properties) of each class. These attributes define the state of the objects.

**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.

**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.

**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).

**Create Objects:**

Instantiate objects from the defined classes. Objects are instances of classes and represent specific entities in the system.

**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.

**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.

**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.

**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.

**Q2: What is the Object Modeling Techniques (OMT).**

is a real-world-based modeling approach used for software modeling and designing.

**Q3: Compare object-oriented analysis and design (OOAD) and object analysis and design (OOP).**

1. **Object-Oriented Analysis and Design (OOAD):**
   * **Analysis:** Involves studying a system to identify its objects, their relationships, and their characteristics. It focuses on understanding the problem domain and defining what needs to be done.
   * **Design:** Involves defining how the identified objects will interact to fulfill the system's requirements. It includes designing classes, relationships, and methods to implement the system.
2. **Object-Oriented Programming (OOP):**
   * **Programming:** Refers to the actual implementation of the system using an object-oriented language like Java, Python, or C++. In OOP, you write code using classes and objects, encapsulation, inheritance, and polymorphism, applying the principles identified during analysis and design.

**Q4: Discuss Mian goals of UML.**

1. **Standardization:**
   * UML aims to establish a standardized modeling language that allows practitioners, analysts, designers, developers, and other stakeholders to communicate and share information seamlessly. The standardization of UML ensures consistency and interoperability across different software development projects.
2. **Visual Representation:**
   * UML provides a graphical notation that allows developers to visually represent various aspects of a software system. This visual representation helps in understanding complex systems more easily than textual descriptions, making it a powerful tool for communication among team members and stakeholders.
3. **Modeling Abstraction:**
   * UML supports different levels of abstraction, allowing modelers to represent a system from various perspectives. This includes high-level conceptual models, detailed design models, and even models representing the implementation. The ability to model at different levels of abstraction contributes to better communication and understanding of complex systems.
4. **Communication and Collaboration:**
   * UML facilitates communication among stakeholders by providing a common language that is not overly technical or tied to a specific programming language. This enables developers, architects, business analysts, and other stakeholders to collaborate effectively, ensuring that everyone involved in the project has a shared understanding of the system.
5. **Documentation:**
   * UML serves as a powerful documentation tool for software systems. It allows developers to document the architecture, design, and implementation details of a system in a visual and standardized way. This documentation is valuable for maintaining, evolving, and transferring knowledge about the system over time.
6. **Flexibility and Extensibility:**
   * UML is designed to be flexible and extensible, allowing practitioners to adapt it to their specific needs. It provides a set of core diagrams (e.g., class diagrams, sequence diagrams) but also allows for the creation of custom diagrams and extensions, making it versatile for different types of systems and domains.
7. **Analysis and Design Support:**
   * UML supports both analysis and design activities in software development. It allows modelers to represent requirements, use cases, classes, relationships, and various other aspects of a system. This makes it a comprehensive tool for capturing and visualizing the entire software development process.

**Q5: DESCRIBE three advantages of using object-oriented to develop an information system**

1. **Modularity for Easier Troubleshooting**:
   * In OOP, you create **classes** that encapsulate both data and functions. When you encounter an issue, you can pinpoint the problem area more efficiently. For instance, if a “car” object breaks down, you know to investigate the “Car” class. This modularity simplifies debugging, as each object operates independently, leaving other components unaffected. [It also enables multiple team members to work simultaneously without duplicating functionality1](https://www.roberthalf.com/us/en/insights/career-development/4-advantages-of-object-oriented-programming).
2. **Reuse of Code Through Inheritance**:
   * Imagine you need to create a “RaceCar” object and a “Limousine” object in addition to your existing “Car” object. Instead of building them from scratch, OOP allows you to define a generic “Car” class. The subclasses (“RaceCar” and “Limousine”) inherit the traits of the generic class while maintaining their unique attributes and functions. [This inheritance technique saves time and promotes code reuse1](https://www.roberthalf.com/us/en/insights/career-development/4-advantages-of-object-oriented-programming).
3. **Enhanced Reusability and Maintainability**:
   * OOP emphasizes reusability by allowing you to reuse existing facilities (classes) rather than recreating them. By creating modular, self-contained objects, you can easily integrate them into larger systems. [Additionally, OOP systems can scale seamlessly from small to large projects, making maintenance more manageable2](https://www.geeksforgeeks.org/benefits-advantages-of-oop/).

**Q6: Briefly explain the following terms as used in object-oriented programming. Write a sample java code to illustrate the implementation of each concept. [12 Marks]**

* 1. **Constructor**
  2. **object**
  3. **Destructor**
  4. **polymorphism**
  5. **class**
  6. **Inheritance**

1. **Constructor**:
   * A **constructor** is a special method within a class that gets called when an object of that class is created. It initializes the object’s state (i.e., sets initial values for its attributes). Constructors have the same name as the class and do not have a return type.
   * Example Java code with a constructor:

class Car {

private String make;

private String model;

// Constructor

public Car (String make, String model) {

this.make = make;

this.model = model;

}

public void displayInfo() {

System.out.println("Make: " + make + ", Model: " + model);

}

}

public class CarDemo {

public static void main(String[] args) {

Car myCar = new Car("Toyota", "Camry");

myCar.displayInfo();

}

}

1. **Object**:
   * An **object** is an instance of a class. It represents a real-world entity or concept. Objects have attributes (data) and methods (functions) associated with them.
   * Example Java code creating an object:

public class ObjectExample {

public static void main(String[] args) {

Car myCar = new Car("Honda", "Civic");

myCar.displayInfo();

}

}

1. **Destructor** (Note: Java does not have explicit destructors):
   * A **destructor** is a method that gets called when an object is destroyed or goes out of scope. In Java, the garbage collector automatically handles memory deallocation, so explicit destructors are not needed.
2. **Polymorphism**:
   * **Polymorphism** allows objects of different classes to be treated as objects of a common superclass. It enables method overriding and dynamic method dispatch.
   * Example Java code showing polymorphism:

class Animal {

public void makeSound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Dog barks");

}

}

public class PolymorphismExample {

public static void main(String[] args) {

Animal myAnimal = new Dog(); // Polymorphism

myAnimal.makeSound(); // Calls Dog's overridden method

}

}

1. **Class**:
   * A **class** is a blueprint or template for creating objects. It defines attributes (fields) and methods that the objects of that class will have.
   * Example Java code defining a class:

class Student {

private String name;

private int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

public void displayInfo() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

1. **Inheritance**:
   * **Inheritance** allows a new class (subclass or derived class) to inherit properties and behaviors from an existing class (superclass or base class). It promotes code reuse.
   * Example Java code demonstrating inheritance:

class Employee {

protected String empId;

public Employee(String empId) {

this.empId = empId;

}

public void displayEmpInfo() {

System.out.println("Employee ID: " + empId);

}

}

class Manager extends Employee {

private String department;

public Manager(String empId, String department) {

super(empId); // Call superclass constructor

this.department = department;

}

public void displayManagerInfo() {

System.out.println("Department: " + department);

}

}

**Q7: *EXPLAIN* the three types of associations (relationships) between objects in object oriented.**

* Association:

Association represents a semantically weak relationship between otherwise unrelated objects. It is a “using” relationship where two or more objects interact without any ownership or parent-child connection.

Example: Consider the relationship between a doctor and a patient. A doctor can be associated with multiple patients, and conversely, a patient can visit multiple doctors. Each object (doctor or patient) has its own lifecycle, and there is no ownership.

Code Example:

public class IDGBlogAccount

{

private IDGBlogEntry[] blogEntries; // Other members of the IDGBlogAccount class

}

public class IDGBlogEntry

{

Int32 blogId;

string caption;

string text;

}

* Aggregation:

Aggregation is a specialized form of association where each object has its own lifecycle, but there exists an ownership relationship. It represents a whole/part or parent/child relationship.

Example: An employee may belong to one or more departments in an organization. If a department is deleted, the employee object continues to exist independently.

Code Example:

public class IDGBlogAuthor

{

// Other members of the IDGBlogAuthor class

}

public class IDGBlogAccount

{

// Other members of the IDGBlogAccount class

private IDGBlogAuthor[] authors; // Aggregation relationship

}

* Composition:

Composition is a stronger form of aggregation. It denotes a strict ownership relationship where the whole (parent) manages the lifecycle of its parts (children).

Example: A car consists of engine, wheels, and other components. If the car is destroyed, its parts are also destroyed.

Key Property: The whole (parent) and its parts (children) have a lifecycle dependency.

Code Example:

public class Car

{

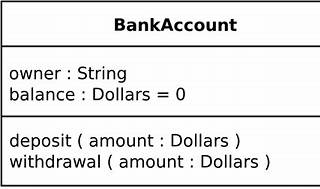
private Engine engine; // Composition relationship

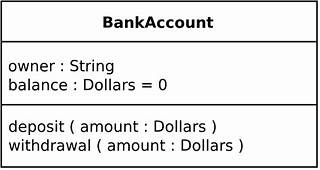
private Wheel[] wheels; // Composition relationship

// Other members of the Car class

}

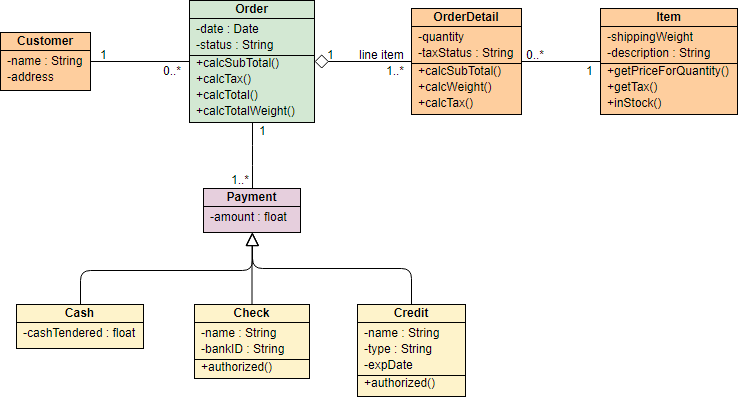
qWhat do you mean by class diagram? Where it is used and also discuss the steps to draw the class diagram with any one example.





1. **What is a Class Diagram?**
   * A **class diagram** is a type of **static structure diagram** in UML. It describes the structure of a system by showing:
     + **Classes**: Representing the building blocks of the system.
     + **Attributes**: Properties or data associated with each class.
     + **Operations (Methods)**: Functions or behaviors that the classes can perform.
     + [**Relationships**: Connections between classes, indicating how they interact with each other](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/)[1](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/).
2. **Purpose and Usage**:
   * **Visualizing Structure**: Class diagrams provide a visual representation of the static structure of a system. They help developers understand the relationships and dependencies among classes.
   * **Notation Foundation**: Class diagrams serve as the foundation for other UML structure diagrams (like component and deployment diagrams).
   * [**Code Generation**: They can be used to construct executable code during forward and reverse engineering of a system](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/)[2](https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/).
3. **Steps to Draw a Class Diagram**:
   * **Identify Classes**:
     + Start by identifying the key classes in your system. These represent the major components or entities.
   * **Add Attributes and Methods**:
     + For each class, list down its attributes (data members) and methods (functions).
   * **Define Relationships**:
     + Determine the relationships between classes:
       - **Association**: Represents a weak relationship between unrelated objects.
       - **Aggregation**: Depicts a whole/part relationship with ownership.
       - **Composition**: Stronger form of aggregation with strict ownership.
   * **Draw the Diagram**:
     + Use UML notation to create the class diagram:
       - Class name in the first partition.
       - Attributes in the second partition (with types).
       - Methods in the third partition (with return types).
     + Connect classes using lines to represent relationships (inheritance, association, etc.).
4. **Example**: library management system:
   * **Classes**:
     + Book: Represents a book with attributes like title, author, and ISBN.
     + Library: Represents a library with attributes like name and location.
   * **Relationships**:
     + Library has an **aggregation** relationship with Book (a library contains multiple books).
     + Book has an **association** relationship with Library (books are associated with a library).

A class diagram example:



Q10: 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. a. Inheritance (Single inheritance, Multiple inheritance and Hierarchical inheritance) [10 Marks] b. Friend functions [5 Marks] c. Method overloading and method overriding [10 Marks] d. Late binding and early binding [8 Marks] e. Abstract class and pure functions

#include <iostream>

#include <cmath>

#ifndef SHAPE\_H

#define SHAPE\_H

#define PI 3.14159265358979323846

class Shape {

public:

virtual double getArea() const = 0;

virtual double getPerimeter() const = 0;

};

class Circle : public Shape {

private:

double radius;

public:

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

double getArea() const override {

return PI \* std::pow(radius, 2);

}

double getPerimeter() const override {

return 2 \* PI \* radius;

}

};

class Rectangle : public Shape {

private:

double length;

double width;

public:

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

double getArea() const override {

return length \* width;

}

double getPerimeter() const override {

return 2 \* (length + width);

}

};

class Triangle : public Shape {

private:

double base;

double height;

public:

Triangle(double b, double h) : base(b), height(h) {}

double getArea() const override {

return 0.5 \* base \* height;

}

double getPerimeter() const override {

return 3 \* base;

}

};

class Square : public Rectangle {

public:

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

};

#endif

int main() {

double radius, length, width, base, height;

std::cout << "Enter the radius of the circle: ";

std::cin >> radius;

Circle circle(radius);

std::cout << "Enter the length and width of the rectangle: ";

std::cin >> length >> width;

Rectangle rectangle(length, width);

std::cout << "Enter the base and height of the triangle: ";

std::cin >> base >> height;

Triangle triangle(base, height);

std::cout << "Enter the side length of the square: ";

std::cin >> length;

Square square(length);

std::cout << "\nCircle: Area = " << circle.getArea() << ", Perimeter = " << circle.getPerimeter() << std::endl;

std::cout << "Rectangle: Area = " << rectangle.getArea() << ", Perimeter = " << rectangle.getPerimeter() << std::endl;

std::cout << "Triangle: Area = " << triangle.getArea() << ", Perimeter = " << triangle.getPerimeter() << std::endl;

std::cout << "Square: Area = " << square.getArea() << ", Perimeter = " << square.getPerimeter() << std::endl;

return 0;

}

**6**

Q11: public class CalculateG {

// Method for multiplication

public double multi(double a, double b) {

return a \* b;

}

// Method for powering to square

public double square(double x) {

return x \* x;

}

// Method for summation

public double sum(double x, double y) {

return x + y;

}

// Method for printing out a result

public void outline(String message, double value) {

System.out.println(message + value);

}

public static void main(String[] args) {

CalculateG calculator = new CalculateG();

double gravity = -9.81; // Earth's gravity in m/s²

double fallingTime = 30.0; // Time in seconds

double initialVelocity = 0.0; // Initial velocity in m/s

double initialPosition = 0.0; // Initial position in meters

// Compute final position

double finalPosition = 0.5 \* calculator.multi(gravity, calculator.square(fallingTime))

+ calculator.multi(initialVelocity, fallingTime) + initialPosition;

// Compute final velocity

double finalVelocity = calculator.sum(calculator.multi(gravity, fallingTime), initialVelocity);

// Output results

calculator.outline("The object's position after " + fallingTime + " seconds is ", finalPosition);

calculator.outline("The object's velocity after " + fallingTime + " seconds is ", finalVelocity);

}

}

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

public class CalculateG {

// multiplication

public double multi(double a, double b) {

return a \* b;

}

//powering to square

public double square(double x) {

return x \* x;

}

// summation

public double sum(double x, double y) {

return x + y;

}

//result

public void outline(String message, double value) {

System.out.println(message + value);

}

public static void main(String[] args) {

CalculateG calculator = new CalculateG();

double gravity = -9.81; // Earth's gravity in m/s²

double fallingTime = 30.0; // Time in seconds

double initialVelocity = 0.0; // Initial velocity in m/s

double initialPosition = 0.0; // Initial position in meters

// Compute final position

double finalPosition = 0.5 \* calculator.multi(gravity, calculator.square(fallingTime))

+ calculator.multi(initialVelocity, fallingTime) + initialPosition;

// Compute final velocity

double finalVelocity = calculator.sum(calculator.multi(gravity, fallingTime), initialVelocity);

// Output results

calculator.outline("The object's position after " + fallingTime + " seconds is ", finalPosition);

calculator.outline("The object's velocity after " + fallingTime + " seconds is ", finalVelocity);

}

}

**PART B:**

**QUESTION 1**

1.Each new term in the Fibonacci sequence is generated by adding the previous two terms. By starting with 1 and 2, the first 10 terms will be:  
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

By considering the terms in the Fibonacci sequence whose values do not exceed four million, write a C++ method to find the sum of all the even- valued terms.

#include <iostream>

using namespace std;

int evenFibSum(int limit) {

if (limit < 2) return 0;

long long int ef1 = 0, ef2 = 2;

long long int sum = ef1 + ef2;

while (ef2 <= limit) {

long long int ef3 = 4 \* ef2 + ef1;

if (ef3 > limit) break;

ef1 = ef2;

ef2 = ef3;

sum += ef2;

}

return sum;

}

int main() {

int limit = 4000000;

cout << evenFibSum(limit);

return 0;

}

(1) Variables and types - C++ Users. https://cplusplus.com/doc/tutorial/variables/.

(2) Declaration and Initialization of Variables: How to Declare ... - Toppr. https://www.toppr.com/guides/computer-science/introduction-to-c/data-types-variables-and-constants/declaration-of-variables/.

(3) Explain the variable declaration, initialization and assignment in C .... https://www.tutorialspoint.com/explain-the-variable-declaration-initialization-and-assignment-in-c-language.

QUESTION 2

Q2#include <QApplication>

#include <QWidget>

#include <QLineEdit>

#include <QPushButton>

#include <QLabel>

#include <QVBoxLayout>

class PalindromeChecker : public QWidget {

Q\_OBJECT

public:

PalindromeChecker(QWidget \*parent = nullptr) : QWidget(parent) {

setupUI();

}

private slots:

void checkPalindrome() {

QString inputText = inputLineEdit->text();

bool isPalindrome = isPalindromeNumber(inputText.toInt());

resultLabel->setText(isPalindrome ? "Palindrome" : "Not a palindrome");

}

private:

QLineEdit \*inputLineEdit;

QLabel \*resultLabel;

bool isPalindromeNumber(int number) {

int originalNumber = number;

int reverseNumber = 0;

while (number > 0) {

int digit = number % 10;

reverseNumber = reverseNumber \* 10 + digit;

number /= 10;

}

return originalNumber == reverseNumber;

}

void setupUI() {

QVBoxLayout \*layout = new QVBoxLayout(this);

inputLineEdit = new QLineEdit(this);

QPushButton \*checkButton = new QPushButton("Check Palindrome", this);

connect(checkButton, &QPushButton::clicked, this, &PalindromeChecker::checkPalindrome);

resultLabel = new QLabel(this);

layout->addWidget(inputLineEdit);

layout->addWidget(checkButton);

layout->addWidget(resultLabel);

setLayout(layout);

setWindowTitle("Palindrome Checker");

}

};

int main(int argc, char \*argv[]) {

QApplication app(argc, argv);

PalindromeChecker palindromeChecker;

palindromeChecker.show();

return app.exec();

}

#include "main.moc"

QUESTION 3

#include <iostream>

int main() {

const int size = 15;

int originalArray[size];

std::cout << "Enter 15 integer values:" << std::endl;

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

std::cout << "Enter value #" << i + 1 << ": ";

std::cin >> originalArray[i];

}

std::cout << "Values stored in the array:" << std::endl;

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

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

}

std::cout << std::endl;

int searchNumber;

std::cout << "Enter 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." << std::endl;

}

int reversedArray[size];

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

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

}

std::cout << "Values in the reversed array:" << std::endl;

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

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

}

std::cout << std::endl;

int sum = 0;

long long product = 1;

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

sum += originalArray[i];

product \*= originalArray[i];

}

std::cout << "Sum of elements: " << sum << std::endl;

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

return 0;

}