CU
CHANDIGARH
UNIVERSITY

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

DAY-1

Student Name: Nikhil **UID:22BCS11017**
Branch: BE-CSE **Section:22BCS_IOT-639-B**
Semester: 5th **Date of Performance: 19/9/2024**
Subject Name: Domain Camp

VERY EASY

1.Aim:

WAP to check the no is prime or not.

2.Objective:

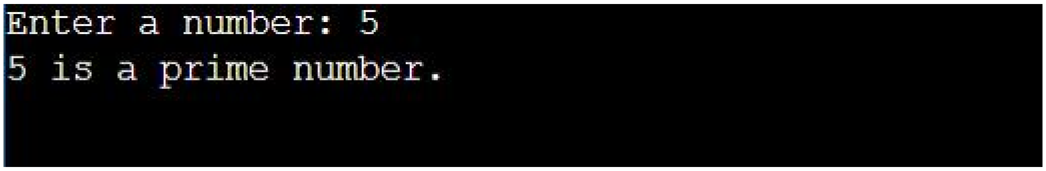
The objective of the program is to determine whether a given number is a prime number or not. A prime number is a natural number greater than 1 that has no divisors other than 1 and itself. The program accomplishes this by taking a user-inputted number and checking if it is divisible by any integer from 2 up to the square root of the number. If a divisor is found, the number is not prime; otherwise, it is classified as prime. This approach ensures computational efficiency while maintaining accuracy, making it suitable for testing both small and large numbers for primality.

3. Implementation of CODE :

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

bool isPrime(int num) {
   if (num <= 1) return false;
   for (int i = 2; i * i <= num; i++) {
      if (num % i == 0) return false;
   }
   return true;
}

int main() {
   int number;
```

```
    cout << "Enter a number: ";
    cin >> number;

    if (isPrime(number)) {
        cout << number << " is a prime number." << endl;
    } else {
        cout << number << " is not a prime number." << endl;
    }

    return 0;
}
```

4.Output:



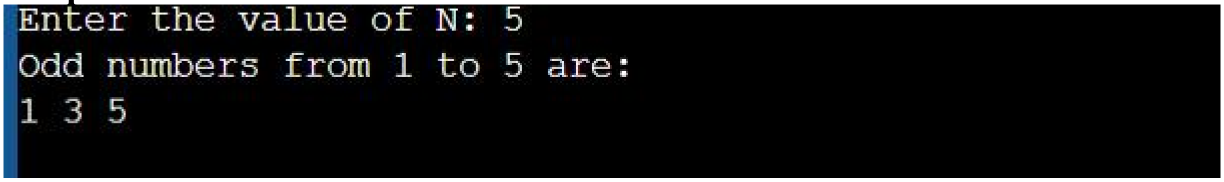
```
Enter a number: 5
5 is a prime number.
```

Aim:WAP print the odd no to N

Code:#include <iostream>

```
using namespace std;

int main() {
    int N;

    cout << "Enter the value of N: ";
    cin >> N;

    cout << "Odd numbers from 1 to " << N << " are:" << endl;
    for (int i = 1; i <= N; i += 2) {
        cout << i << " ";
    }

    cout << endl;
    return 0;
```

Output:



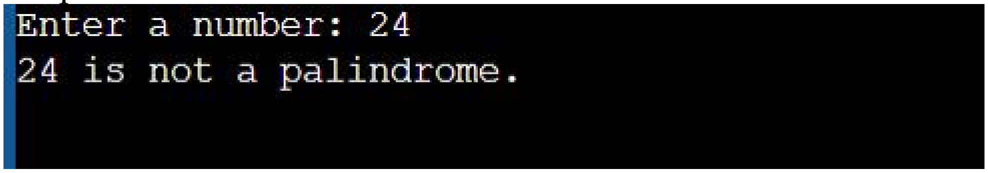
```
Enter the value of N: 5
Odd numbers from 1 to 5 are:
1 3 5
```

Aim:WAP check the no is pallindrom or not.

Code: #include <iostream>

```
using namespace std;

int main() {
    int num, reversedNum = 0, remainder, originalNum;

    cout << "Enter a number: ";
    cin >> num;

    originalNum = num;

    while (num != 0) {
        remainder = num % 10;
        reversedNum = reversedNum * 10 + remainder;
        num /= 10;
    }

    if (originalNum == reversedNum) {
        cout << originalNum << " is a palindrome." << endl;
    } else {
        cout << originalNum << " is not a palindrome." << endl;
    }

    return 0;
}
```

Output:



```
Enter a number: 24
24 is not a palindrome.
```

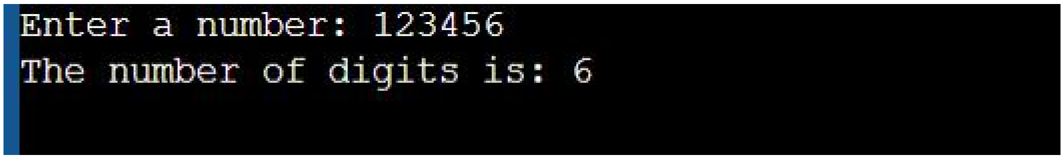
Easy

Aim: WAP count the digit of the no.

Code:

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

int main() {
    int num, count = 0;

    cout << "Enter a number: ";
    cin >> num;

    if (num == 0) {
        count = 1;
    } else {
        while (num != 0) {
            num /= 10;
            count++;
        }
    }

    cout << "The number of digits is: " << count << endl;

    return 0;
}
```

Output:



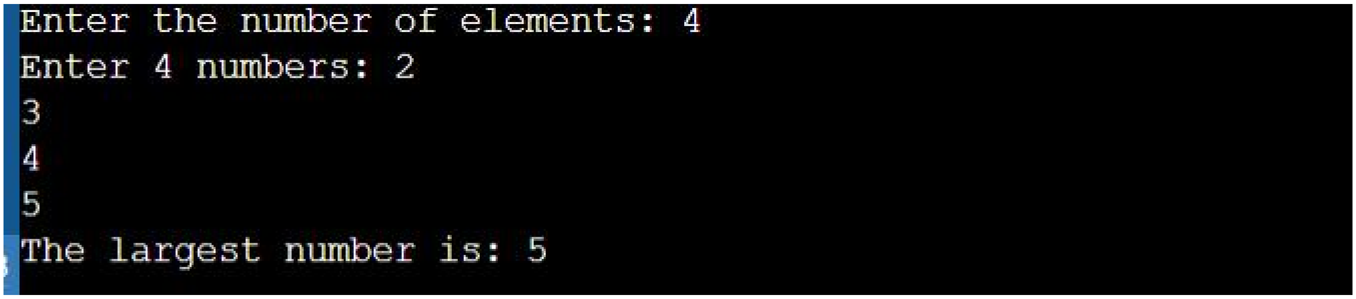
```
Enter a number: 123456
The number of digits is: 6
```

Aim: WAP find the largest no.

Code:

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

int main() {
    int n;

    cout << "Enter the number of elements: ";
    cin >> n;

    int arr[n];

    cout << "Enter " << n << " numbers: ";
    for (int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int largest = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > largest) {
            largest = arr[i];
        }
    }

    cout << "The largest number is: " << largest << endl;

    return 0;
}
```

Output:



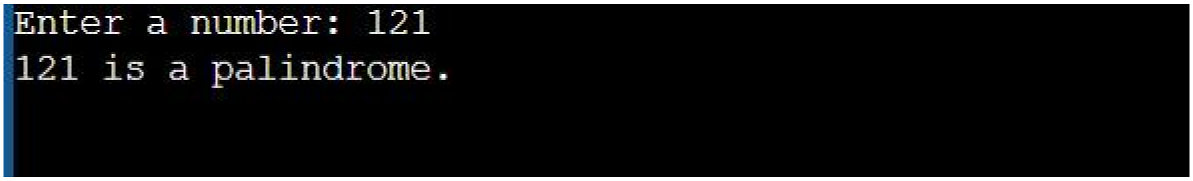
```
Enter the number of elements: 4
Enter 4 numbers: 2
3
4
5
The largest number is: 5
```

Aim: WAP fine the no id pallindrom or not.

Code:

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

int main() {
    int num, originalNum, reversedNum = 0, remainder;

    cout << "Enter a number: ";
    cin >> num;

    originalNum = num;

    while (num != 0) {
        remainder = num % 10;
        reversedNum = reversedNum * 10 + remainder;
        num /= 10;
    }

    if (originalNum == reversedNum) {
        cout << originalNum << " is a palindrome." << endl;
    } else {
        cout << originalNum << " is not a palindrome." << endl;
    }

    return 0;
}
```

Output:



```
Enter a number: 121
121 is a palindrome.
```

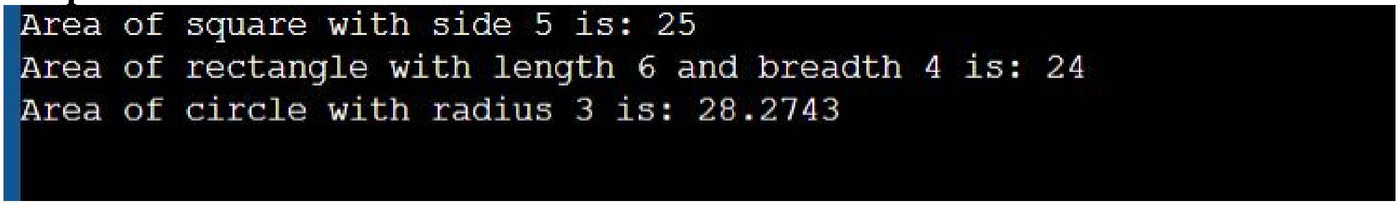
MEDIUM

Aim: WAP function overloading for calculating the area.

Code:

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

double area(double side) {
    return side * side;
}
double area(double length, double breadth) {
    return length * breadth;
}
double area(double radius, bool isCircle) {
    const double PI = 3.14159;
    return PI * radius * radius;
}
int main() {
    double squareSide = 5.0;
    cout << "Area of square with side " << squareSide << " is: " << area(squareSide) << endl;

    double rectangleLength = 6.0, rectangleBreadth = 4.0;
    cout << "Area of rectangle with length " << rectangleLength
         << " and breadth " << rectangleBreadth << " is: "
         << area(rectangleLength, rectangleBreadth) << endl;

     double circleRadius = 3.0;
    cout << "Area of circle with radius " << circleRadius << " is: "
         << area(circleRadius, true) << endl;

    return 0;
}
```

Output:



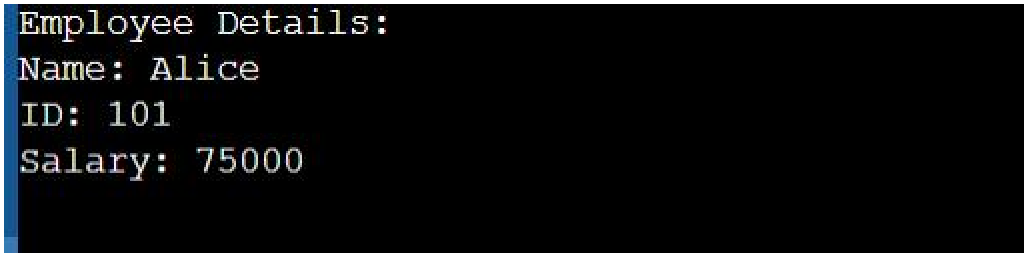
```
Area of square with side 5 is: 25
Area of rectangle with length 6 and breadth 4 is: 24
Area of circle with radius 3 is: 28.2743
```

Aim: WAP encapsulation using employee details.

Code:

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

class Employee {
private:
    string name;
    int id;
    double salary;

public:

    void setName(string empName) {
        name = empName;
    }

    string getName() const {
        return name;
    }

    void setId(int empId) {
        if (empId > 0) {
            id = empId;
        } else {
            cout << "Invalid ID!" << endl;
        }
    }

    int getId() const {
        return id;
    }

    void setSalary(double empSalary) {
        if (empSalary >= 0) {
            salary = empSalary;
        } else {
            cout << "Invalid salary!" << endl;
        }
    }

    double getSalary() const {
        return salary;
    }

    void displayDetails() const {
        cout << "Employee Details:\n";
        cout << "Name: " << name << "\nID: " << id << "\nSalary: " << salary << endl;
    }
};

int main() {
    Employee emp;

    emp.setName("Alice");
    emp.setId(101);
    emp.setSalary(75000.00);

    cout << "Accessing Employee Details Using Getters:\n";
    cout << "Name: " << emp.getName() << endl;
    cout << "ID: " << emp.getId() << endl;
    cout << "Salary: " << emp.getSalary() << endl;

    cout << "\nDisplaying Employee Details Using Display Function:\n";
    emp.displayDetails();

    return 0;
}
```

Output:



```
Employee Details:
Name: Alice
ID: 101
Salary: 75000
```

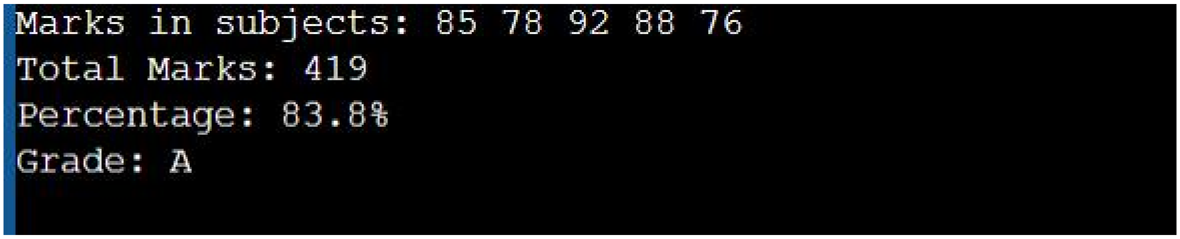
Aim: WAP Inheritance using student& result of class.

Code:

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

class Student {
protected:
    string name;
    int rollNumber;
    int marks[5];

public:
    Student(string studentName, int rollNo, int marksArray[]) {
        name = studentName;
        rollNumber = rollNo;
        for (int i = 0; i < 5; i++) {
            marks[i] = marksArray[i];
        }
    }

    void displayStudentDetails() {
        cout << "Name: " << name << endl;
        cout << "Roll Number: " << rollNumber << endl;
        cout << "Marks in subjects: ";
        for (int i = 0; i < 5; i++) {
            cout << marks[i] << " ";
        }
        cout << endl;
    }
};

class Result : public Student {
private:
    double totalMarks;
    double percentage;

public:
    Result(string studentName, int rollNo, int marksArray[])
```

```
        : Student(studentName, rollNo, marksArray) {
        calculateResult();
    }

    void calculateResult() {
        totalMarks = 0;
        for (int i = 0; i < 5; i++) {
            totalMarks += marks[i];
        }
        percentage = totalMarks / 5.0;
    }

    void displayResult() {
        displayStudentDetails();
        cout << "Total Marks: " << totalMarks << endl;
        cout << "Percentage: " << percentage << "%" << endl;

        if (percentage >= 90) {
            cout << "Grade: A+" << endl;
        } else if (percentage >= 80) {
            cout << "Grade: A" << endl;
        } else if (percentage >= 70) {
            cout << "Grade: B+" << endl;
        } else if (percentage >= 60) {
            cout << "Grade: B" << endl;
        } else if (percentage >= 50) {
            cout << "Grade: C" << endl;
        } else {
            cout << "Grade: F" << endl;
        }
    }
};

int main() {
    int marksArray[5] = {85, 78, 92, 88, 76};
    Result studentResult("John Doe", 101, marksArray);

    studentResult.displayResult();

    return 0;
}
```

Output:



```
Marks in subjects: 85 78 92 88 76
Total Marks: 419
Percentage: 83.8%
Grade: A
```

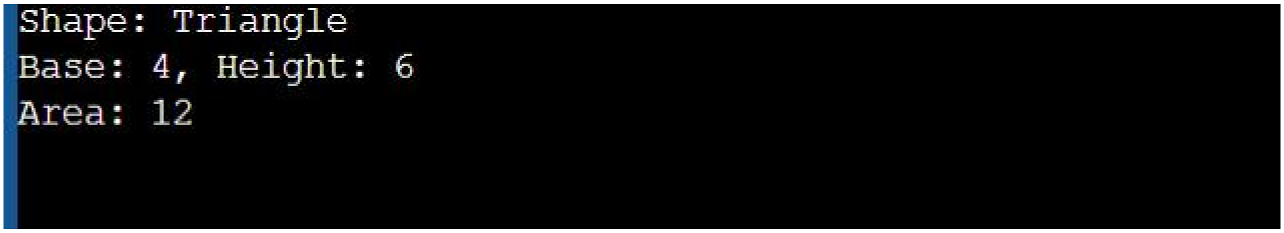
HARD

Aim: WAP implementing polymorphism for shape hierarchies.

Code:

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

class Shape {
public:
    virtual double area() = 0;
    virtual void display() = 0;
};

class Circle : public Shape {
private:
    double radius;

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

    double area() override {
        return M_PI * radius * radius;
    }

    void display() override {
        cout << "Shape: Circle" << endl;
        cout << "Radius: " << radius << endl;
        cout << "Area: " << area() << endl;
    }
};

class Rectangle : public Shape {
private:
    double length, width;

public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double area() override {
        return length * width;
    }

    void display() override {
        cout << "Shape: Rectangle" << endl;
        cout << "Length: " << length << ", Width: " << width << endl;
        cout << "Area: " << area() << endl;
    }
};

class Triangle : public Shape {
private:
    double base, height;

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

    double area() override {
        return 0.5 * base * height;
    }

    void display() override {
        cout << "Shape: Triangle" << endl;
        cout << "Base: " << base << ", Height: " << height << endl;
        cout << "Area: " << area() << endl;
    }
};

int main() {
    Shape* shape1 = new Circle(5.0);
    Shape* shape2 = new Rectangle(4.0, 6.0);
    Shape* shape3 = new Triangle(4.0, 6.0);

    Shape* shapes[] = {shape1, shape2, shape3};

    for (int i = 0; i < 3; i++) {
        shapes[i]->display();
        cout << endl;
    }

    delete shape1;
    delete shape2;
    delete shape3;

    return 0;
}
```

Output:



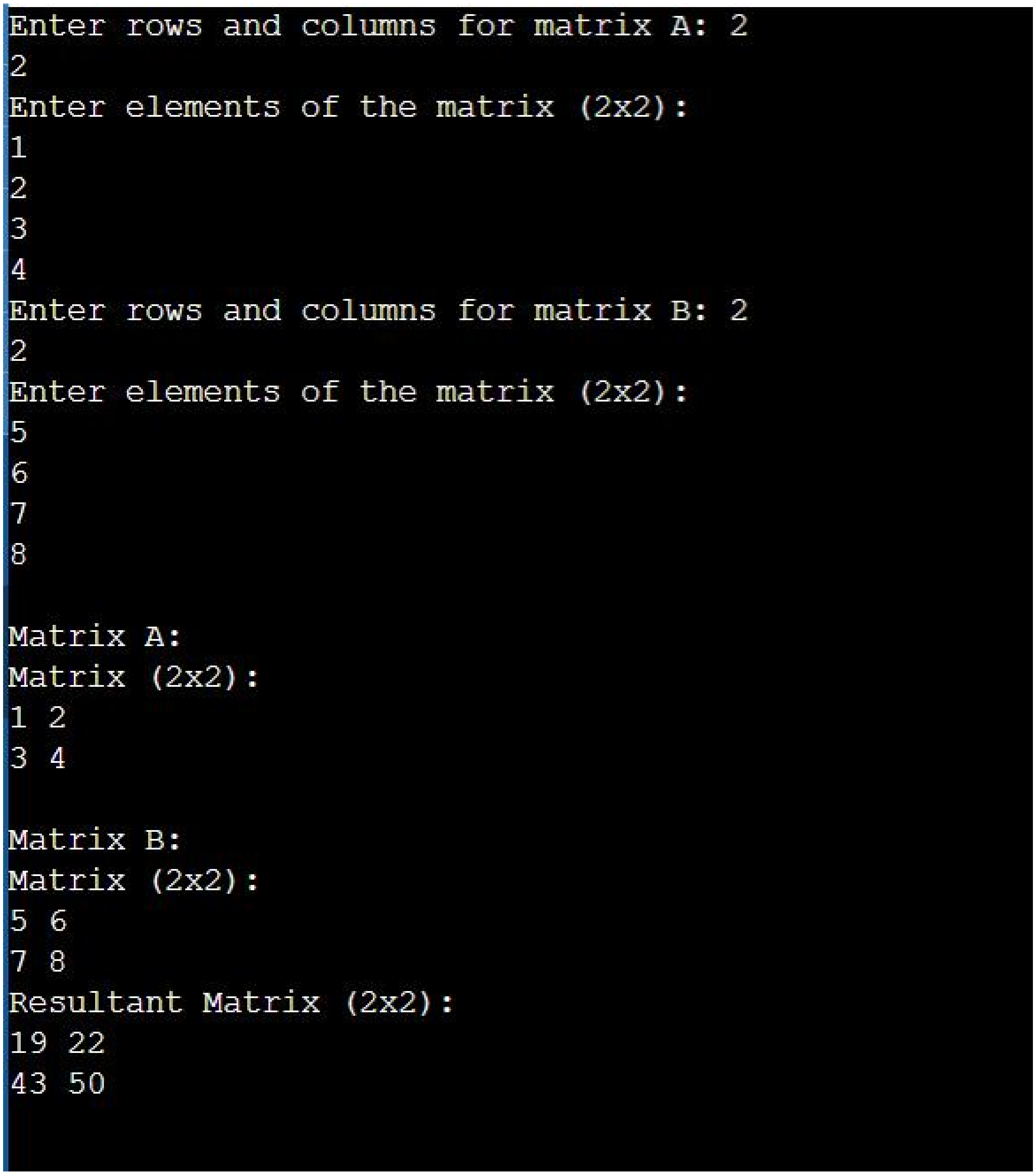
```
Shape: Triangle
Base: 4, Height: 6
Area: 12
```

Aim: WAP matrix multiplication using function overloading.

Code:

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

class Matrix {
private:
    int mat[10][10];

public:
    void inputMatrix(int row, int col) {
        cout << "Enter elements of the matrix (" << row << "x" << col << "):\n";
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                cin >> mat[i][j];
            }
        }
    }

    void displayMatrix(int row, int col) {
        cout << "Matrix (" << row << "x" << col << "):\n";
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                cout << mat[i][j] << " ";
            }
            cout << endl;
        }
    }

    void multiply(int A[10][10], int B[10][10], int A_row, int A_col, int B_row, int B_col, int result[10][10]) {
        if (A_col != B_row) {
            cout << "Matrix multiplication not possible. Column of A must be equal to row of B.\n";
            return;
        }

        for (int i = 0; i < A_row; i++) {
            for (int j = 0; j < B_col; j++) {
                result[i][j] = 0;
                for (int k = 0; k < A_col; k++) {
                    result[i][j] += A[i][k] * B[k][j];
                }
            }
        }
    }

    void multiply(Matrix m1, Matrix m2, int A_row, int A_col, int B_row, int B_col, int result[10][10]) {
        if (A_col != B_row) {
            cout << "Matrix multiplication not possible. Column of A must be equal to row of B.\n";
            return;
        }

        for (int i = 0; i < A_row; i++) {
            for (int j = 0; j < B_col; j++) {
                result[i][j] = 0;
                for (int k = 0; k < A_col; k++) {
                    result[i][j] += m1.mat[i][k] * m2.mat[k][j];
                }
            }
        }
    }

    void displayResult(int result[10][10], int row, int col) {
        cout << "Resultant Matrix (" << row << "x" << col << "):\n";
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                cout << result[i][j] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    Matrix m1, m2;
    int A_row, A_col, B_row, B_col;
    int result[10][10];

    cout << "Enter rows and columns for matrix A: ";
    cin >> A_row >> A_col;
    m1.inputMatrix(A_row, A_col);

    cout << "Enter rows and columns for matrix B: ";
    cin >> B_row >> B_col;
    m2.inputMatrix(B_row, B_col);

    cout << "\nMatrix A:\n";
    m1.displayMatrix(A_row, A_col);
    cout << "\nMatrix B:\n";
    m2.displayMatrix(B_row, B_col);

    m1.multiply(m1, m2, A_row, A_col, B_row, B_col, result);

    m1.displayResult(result, A_row, B_col);

    return 0;
}
```

Output:



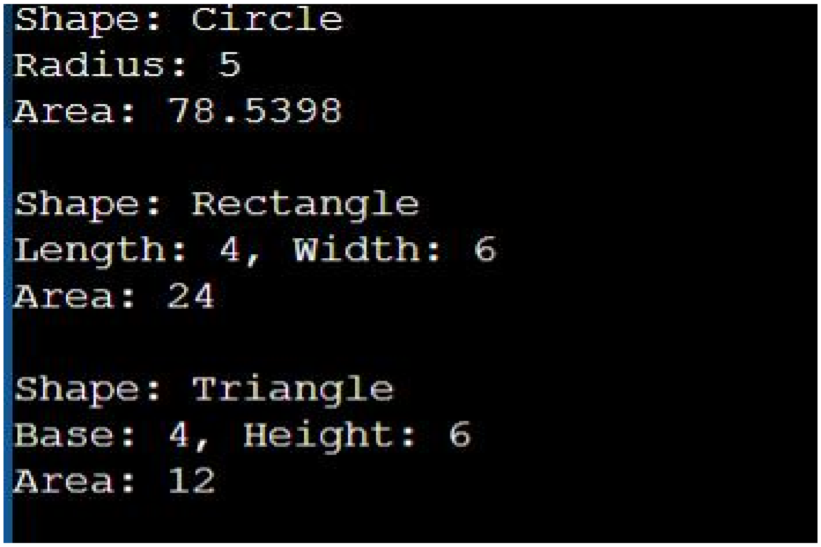
```
Enter rows and columns for matrix A: 2
2
Enter elements of the matrix (2x2):
1
2
3
4
Enter rows and columns for matrix B: 2
2
Enter elements of the matrix (2x2):
5
6
7
8

Matrix A:
Matrix (2x2):
1 2
3 4

Matrix B:
Matrix (2x2):
5 6
7 8
Resultant Matrix (2x2):
19 22
43 50
```

Aim:WAP polymorphism in shape classes.

Code:

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

class Shape {
public:
    virtual double area() const = 0;

    virtual void display() const = 0;

    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

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

    double area() const override {
        return M_PI * radius * radius;
    }

    void display() const override {
        cout << "Shape: Circle" << endl;
        cout << "Radius: " << radius << endl;
        cout << "Area: " << area() << endl;
    }
};

class Rectangle : public Shape {
private:
    double length, width;

public:
    Rectangle(double l, double w) : length(l), width(w) {}
    double area() const override {
        return length * width;
    }

    void display() const override {
        cout << "Shape: Rectangle" << endl;
        cout << "Length: " << length << ", Width: " << width << endl;
        cout << "Area: " << area() << endl;
    }
};

class Triangle : public Shape {
private:
    double base, height;

public:

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

    double area() const override {
        return 0.5 * base * height;
    }

    void display() const override {
        cout << "Shape: Triangle" << endl;
        cout << "Base: " << base << ", Height: " << height << endl;
        cout << "Area: " << area() << endl;
    }
};

int main() {
    Shape* shape1 = new Circle(5.0);
    Shape* shape2 = new Rectangle(4.0, 6.0);
    Shape* shape3 = new Triangle(4.0, 6.0);

    Shape* shapes[] = {shape1, shape2, shape3};

    for (int i = 0; i < 3; i++) {
        shapes[i]->display();
        cout << endl;
    }
    delete shape1;
    delete shape2;
    delete shape3;

    return 0;
}
```

Output:



```
Shape: Circle
Radius: 5
Area: 78.5398

Shape: Rectangle
Length: 4, Width: 6
Area: 24

Shape: Triangle
Base: 4, Height: 6
Area: 12
```

VERY HARD

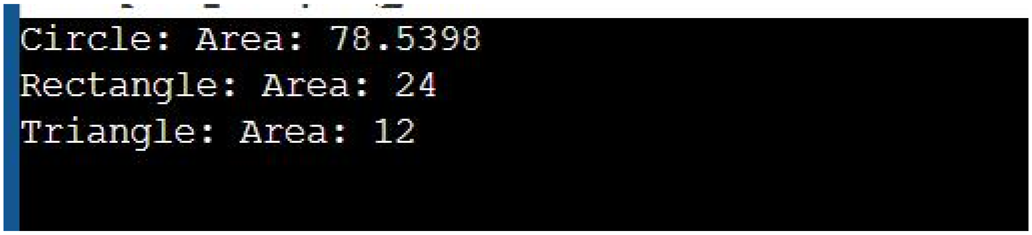
Aim: WAP polymorphism for shape area calculation.

Code:

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

class Shape {
public:
    virtual double area() const = 0;

    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

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

    double area() const override {
        return M_PI * radius * radius;
    }
};

class Rectangle : public Shape {
private:
    double length, width;

public:

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

    double area() const override {
        return length * width;
    }
};
```

```
class Triangle : public Shape {
private:
    double base, height;

public:

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

    double area() const override {
        return 0.5 * base * height;
    }
};

void displayArea(Shape* shape) {
    cout << "Area: " << shape->area() << endl;
}

int main() {
    Shape* circle = new Circle(5.0);
    Shape* rectangle = new Rectangle(4.0, 6.0);
    Shape* triangle = new Triangle(4.0, 6.0);

    cout << "Circle: ";
    displayArea(circle);

    cout << "Rectangle: ";
    displayArea(rectangle);

    cout << "Triangle: ";
    displayArea(triangle);

    delete circle;
    delete rectangle;
    delete triangle;

    return 0;
}
```

Output:



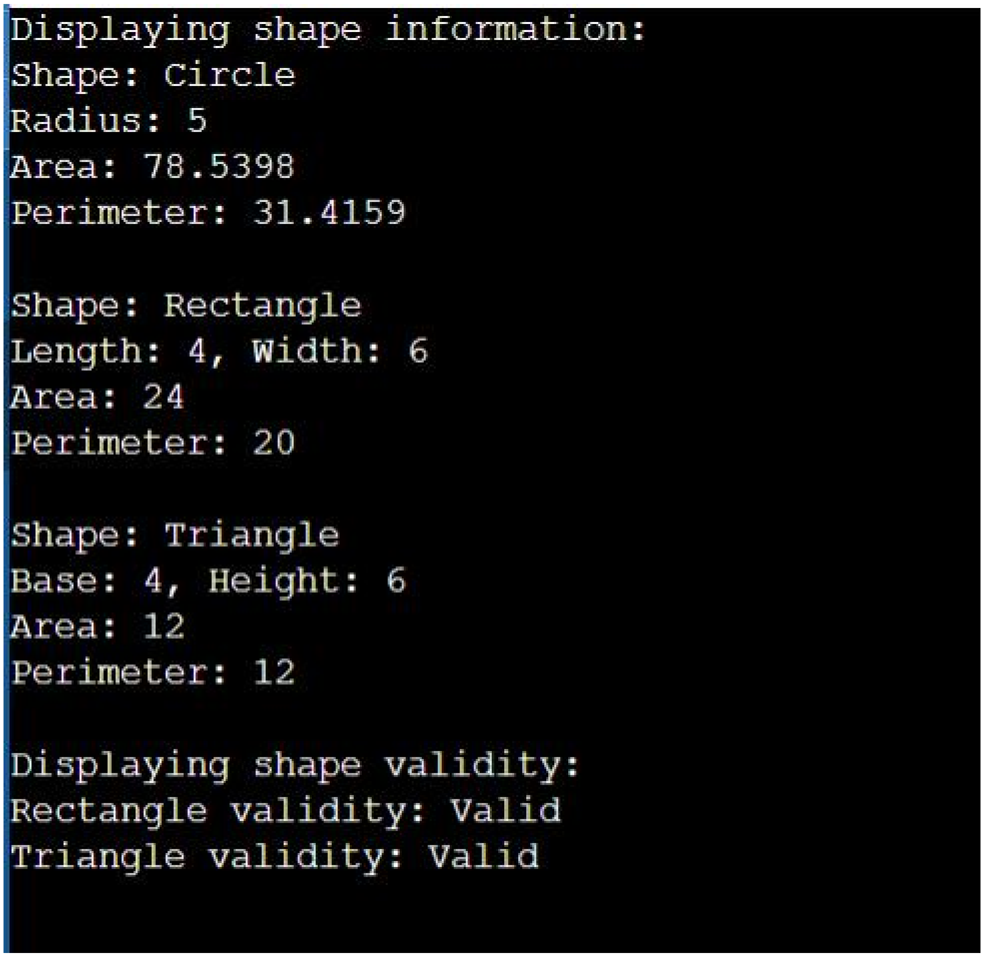
```
Circle: Area: 78.5398
Rectangle: Area: 24
Triangle: Area: 12
```

Aim: WAP advanced function overloading for geometric shapes.

Code:

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

class Shape {
public:
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual void display() const = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

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

    double area() const override {
        return M_PI * radius * radius;
    }

    double perimeter() const override {
        return 2 * M_PI * radius;
    }

    void display() const override {
        cout << "Shape: Circle\n";
        cout << "Radius: " << radius << "\n";
        cout << "Area: " << area() << "\n";
        cout << "Perimeter: " << perimeter() << "\n\n";
    }
};

class Rectangle : public Shape {
private:
    double length, width;

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

    double area() const override {
        return length * width;
    }

    double perimeter() const override {
        return 2 * (length + width);
    }

    void display() const override {
        cout << "Shape: Rectangle\n";
        cout << "Length: " << length << ", Width: " << width << "\n";
        cout << "Area: " << area() << "\n";
        cout << "Perimeter: " << perimeter() << "\n\n";
    }

    bool isValid() const {
        return (length > 0 && width > 0);
    }
};

class Triangle : public Shape {
private:
    double base, height;

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

    double area() const override {
        return 0.5 * base * height;
    }

    double perimeter() const override {
        return 3 * base;
    }

    void display() const override {
        cout << "Shape: Triangle\n";
        cout << "Base: " << base << ", Height: " << height << "\n";
        cout << "Area: " << area() << "\n";
        cout << "Perimeter: " << perimeter() << "\n\n";
    }

    bool isValid() const {
        return (base > 0 && height > 0);
    }
};

void displayShapeInfo(Shape* shape) {
    shape->display();
}

void displayValidity(Rectangle* rect) {
    cout << "Rectangle validity: " << (rect->isValid() ? "Valid" : "Invalid") << "\n";
}

void displayValidity(Triangle* tri) {
    cout << "Triangle validity: " << (tri->isValid() ? "Valid" : "Invalid") << "\n";
}

int main() {
    Shape* circle = new Circle(5.0);
    Shape* rectangle = new Rectangle(4.0, 6.0);
    Shape* triangle = new Triangle(4.0, 6.0);

    cout << "Displaying shape information:\n";
    displayShapeInfo(circle);
    displayShapeInfo(rectangle);
    displayShapeInfo(triangle);

    cout << "Displaying shape validity:\n";
    displayValidity(dynamic_cast<Rectangle*>(rectangle));
    displayValidity(dynamic_cast<Triangle*>(triangle));

    delete circle;
    delete rectangle;
    delete triangle;

    return 0;
}
```

Output:



```
Displaying shape information:
Shape: Circle
Radius: 5
Area: 78.5398
Perimeter: 31.4159

Shape: Rectangle
Length: 4, Width: 6
Area: 24
Perimeter: 20

Shape: Triangle
Base: 4, Height: 6
Area: 12
Perimeter: 12

Displaying shape validity:
Rectangle validity: Valid
Triangle validity: Valid
```