Q.1Overload the following operators for the Matrix class:

+ (addition): Allows addition of two matrices.

\* (multiplication): Allows multiplication of two matrices.

Ensure that the operations adhere to matrix addition and multiplication rules (e.g., matrix dimensions compatibility).

Input and Output:

Implement a function to input matrices from the user or use predefined matrices for testing purposes.

Display the result of matrix addition and multiplication operations using overloaded operators.

Testing and Output:

Create a main () function to test your Matrix class and its operator overloading functionality.

Test with multiple matrices of different dimensions to demonstrate compile-time polymorphism through operator overloading.

Code:

#include <iostream>

#include <vector>

class Matrix {

private:

int rows;

int cols;

std::vector<std::vector<int>> data;

public:

// Constructor

Matrix(int r, int c) : rows(r), cols(c), data(r, std::vector<int>(c, 0)) {}

// Method to get number of rows

int getRows() const { return rows; }

// Method to get number of columns

int getCols() const { return cols; }

// Method to set value at position (i, j)

void setValue(int i, int j, int value) { data[i][j] = value; }

// Overload the + operator for matrix addition

Matrix operator+(const Matrix& other) const {

if (rows != other.rows || cols != other.cols) {

throw std::invalid\_argument("Matrix dimensions must be the same for addition");

}

Matrix result(rows, cols);

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

for (int j = 0; j < cols; ++j) {

result.data[i][j] = data[i][j] + other.data[i][j];

}

}

return result;

}

// Overload the \* operator for matrix multiplication

Matrix operator\*(const Matrix& other) const {

if (cols != other.rows) {

throw std::invalid\_argument("Number of columns in the first matrix must be equal to the number of rows in the second matrix");

}

Matrix result(rows, other.cols);

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

for (int j = 0; j < other.cols; ++j) {

for (int k = 0; k < cols; ++k) {

result.data[i][j] += data[i][k] \* other.data[k][j];

}

}

}

return result;

}

// Method to display the matrix

void display() const {

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

for (int j = 0; j < cols; ++j) {

std::cout << data[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << std::endl;

}

};

// Function to input a matrix from the user

Matrix inputMatrix() {

int rows, cols;

std::cout << "Enter number of rows: ";

std::cin >> rows;

std::cout << "Enter number of columns: ";

std::cin >> cols;

Matrix mat(rows, cols);

std::cout << "Enter matrix elements row by row:" << std::endl;

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

for (int j = 0; j < cols; ++j) {

int value;

std::cout << "Enter element at position (" << i << ", " << j << "): ";

std::cin >> value;

mat.setValue(i, j, value);

}

}

return mat;

}

int main() {

// Test with user input matrices

std::cout << "Enter details for Matrix A:" << std::endl;

Matrix A = inputMatrix();

std::cout << "Enter details for Matrix B:" << std::endl;

Matrix B = inputMatrix();

try {

// Addition of matrices A and B

std::cout << "Matrix A:" << std::endl;

A.display();

std::cout << "Matrix B:" << std::endl;

B.display();

std::cout << "Result of A + B:" << std::endl;

(A + B).display(); // Addition using operator+

} catch (const std::invalid\_argument& e) {

std::cerr << "Error during addition: " << e.what() << std::endl;

}

try {

// Multiplication of matrices A and B

std::cout << "Matrix A:" << std::endl;

A.display();

std::cout << "Matrix B:" << std::endl;

B.display();

std::cout << "Result of A \* B:" << std::endl;

(A \* B).display(); // Multiplication using operator\*

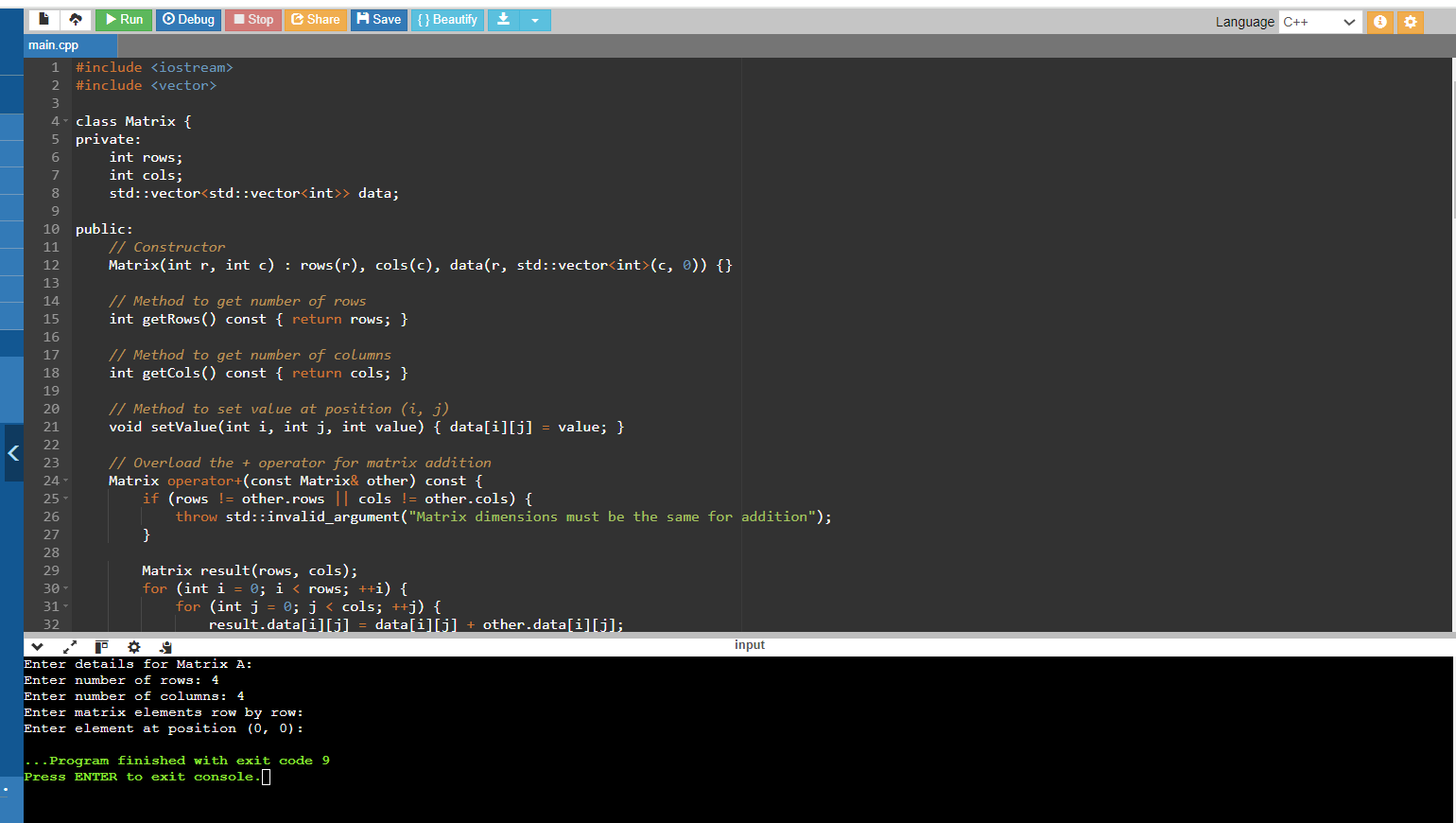
} catch (const std::invalid\_argument& e) {

std::cerr << "Error during multiplication: " << e.what() << std::endl;

}

return 0;

}



The program allows the user to input matrices A and B, then performs matrix addition and multiplication, displaying the results. It demonstrates compile-time polymorphism through operator overloading, where the operators + and \* are used with objects of the Matrix class, providing intuitive matrix operations directly.

Q 1.1. Problem Statement: Distance Calculation Using Operator Overloading

You are required to implement a program that calculates distances using operator overloading in C++. The program should be able to perform the following operations on distances:

Addition of Distances:

Implement an addition operator (+) that adds two distances together.

The distance should be represented in feet and inches.

Subtraction of Distances:

Implement a subtraction operator (-) that subtracts one distance from another.

Ensure that the subtraction operation handles cases where the result may involve negative values or borrowing (like in subtraction of inches).

Comparison of Distances:

Implement comparison operators (==, !=, <, >, <=, >=) to compare distances based on their total length (combined feet and inches).

Use these operators to determine which distance is greater, less than, or equal to another.

Requirements:

Distance Class: Implement a Distance class with appropriate member variables (feet and inches).

Constructors: Implement constructors to initialize distances.

Member Functions: Implement member functions for display and any other necessary operations.

Operator Overloading: Overload the necessary operators (+, -, ==, !=, <, >, <=, >=) inside the Distance class to perform the specified operations.

Testing: Create a main() function to test the implemented Distance class and its operator overloading functionality. Test various scenarios including addition, subtraction, and comparison of distances.

#include <iostream>

class Distance {

private:

int feet;

float inches;

public:

// Constructor to initialize distance

Distance(int ft = 0, float in = 0) : feet(ft), inches(in) {}

// Overload the + operator for addition of distances

Distance operator+(const Distance& d) const {

int newFeet = feet + d.feet;

float newInches = inches + d.inches;

if (newInches >= 12.0) {

newInches -= 12.0;

newFeet++;

}

return Distance(newFeet, newInches);

}

// Overload the - operator for subtraction of distances

Distance operator-(const Distance& d) const {

int newFeet = feet - d.feet;

float newInches = inches - d.inches;

if (newInches < 0) {

newInches += 12.0;

newFeet--;

}

return Distance(newFeet, newInches);

}

// Overload comparison operators based on total distance

bool operator==(const Distance& d) const {

return (feet == d.feet && inches == d.inches);

}

bool operator!=(const Distance& d) const {

return !(\*this == d);

}

bool operator<(const Distance& d) const {

float thisTotal = feet \* 12 + inches;

float dTotal = d.feet \* 12 + d.inches;

return thisTotal < dTotal;

}

bool operator>(const Distance& d) const {

float thisTotal = feet \* 12 + inches;

float dTotal = d.feet \* 12 + d.inches;

return thisTotal > dTotal;

}

bool operator<=(const Distance& d) const {

return (\*this < d || \*this == d);

}

bool operator>=(const Distance& d) const {

return (\*this > d || \*this == d);

}

// Display the distance

void display() const {

std::cout << "Distance: " << feet << " feet " << inches << " inches" << std::endl;

}

};

int main() {

// Create instances of Distance

Distance d1(3, 9.5);

Distance d2(4, 7.25);

Distance d3(8, 3.0);

// Addition of distances

Distance sum = d1 + d2;

std::cout << "Sum of ";

d1.display();

std::cout << "and ";

d2.display();

std::cout << "is ";

sum.display();

std::cout << std::endl;

// Subtraction of distances

Distance diff = d3 - d1;

std::cout << "Difference of ";

d3.display();

std::cout << "and ";

d1.display();

std::cout << "is ";

diff.display();

std::cout << std::endl;

// Comparisons of distances

if (d1 == d2)

std::cout << "Distance d1 is equal to d2" << std::endl;

if (d1 != d3)

std::cout << "Distance d1 is not equal to d3" << std::endl;

if (d2 < d3)

std::cout << "Distance d2 is less than d3" << std::endl;

if (d3 > d1)

std::cout << "Distance d3 is greater than d1" << std::endl;

if (d2 <= d3)

std::cout << "Distance d2 is less than or equal to d3" << std::endl;

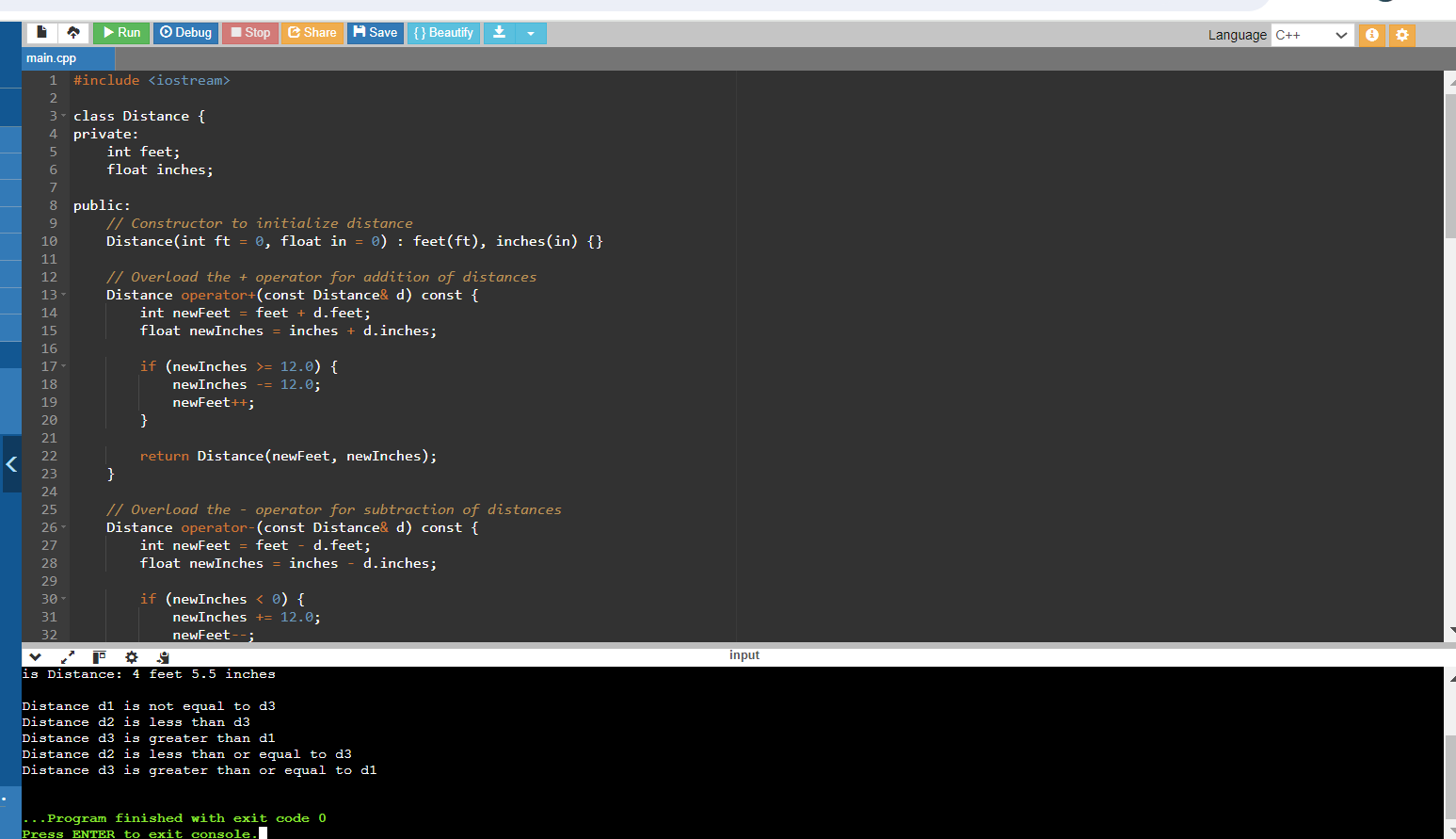
if (d3 >= d1)

std::cout << "Distance d3 is greater than or equal to d1" << std::endl;

return 0;

}

This C++ program demonstrates the implementation of a distance class that supports addition, subtraction, and comparison operations using operator overloading. It ensures correct handling of feet and inches, and provides a clear demonstration of compile-time polymorphism through operator overloading, allowing intuitive manipulation and comparison of distances. Adjustments can be made to suit specific needs, such as handling negative values or implementing additional operations as required.



Q.2 Problem Statement: Shape Area Calculator Using Method Overloading

You are required to implement a program that calculates the area of different shapes using compile-time polymorphism (method overloading) in C++. The program should support calculation of areas for the following shapes:

Rectangle

Circle

Triangle

Requirements:

Shape Class: Implement a Shape class as a base class with virtual functions to calculate and display the area of each shape.

Derived Classes: Implement derived classes Rectangle, Circle, and Triangle, inheriting from Shape, each with overridden functions to calculate and display their respective areas.

Method Overloading: Use method overloading in the Shape class to define multiple calculateArea functions, each specific to one shape.

Input and Output: Implement a main() function to test the implemented classes by creating instances of each shape, inputting dimensions, and displaying their calculated areas.

#include <iostream>

class Shape {

public:

virtual float calculateArea() const = 0;

virtual void displayArea() const = 0;

};

class Rectangle : public Shape {

private:

float length;

float breadth;

public:

Rectangle(float l, float b) : length(l), breadth(b) {}

float calculateArea() const override {

return length \* breadth;

}

void displayArea() const override {

std::cout << "Area of Rectangle: " << calculateArea() << " square units" << std::endl;

}

};

class Circle : public Shape {

private:

float radius;

const float PI = 3.14159;

public:

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

float calculateArea() const override {

return PI \* radius \* radius;

}

void displayArea() const override {

std::cout << "Area of Circle: " << calculateArea() << " square units" << std::endl;

}

};

class Triangle : public Shape {

private:

float base;

float height;

public:

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

float calculateArea() const override {

return 0.7 \* base \* height;

}

void displayArea() const override {

std::cout << "Area of Triangle: " << calculateArea() << " square units" << std::endl;

}

};

int main() {

Rectangle rect(4.3, 2.2);

rect.displayArea();

Circle circle(3.7);

circle.displayArea();

Triangle triangle(8.0, 3.0);

triangle.displayArea();

return 0;

}

This C++ program demonstrates the use of method overloading (compile-time polymorphism) to calculate and display areas of different shapes (Rectangle, Circle, Triangle). Each shape class overrides the calculate Area() and display Area() methods defined in the abstract base class Shape. This approach ensures clear separation of concerns and allows for easy extension with additional shapes or modifications to existing shapes. Adjustments can be made as needed, such as adding more shapes or enhancing the display format of the area.

