Programming in C and C++

Telmen Enkhbold

San Francisco Bay University

CS360

Mr. Yang, Alex

2/14/2024

1.

#include <iostream>

#include <cmath>

#include <sstream>

class Complex {

private:

    double real;

    double imaginary;

public:

    // Default constructor

    Complex() : real(0), imaginary(0) {}

    // Constructor with real and imaginary parts

    Complex(double real, double imaginary) : real(real), imaginary(imaginary) {}

    // Constructor from string

    Complex(const std::string& complexString) {

        std::istringstream iss(complexString);

        char discard;

        iss >> discard >> real >> discard >> imaginary >> discard;

    }

    // Getter for real part

    double getReal() const { return real; }

    // Getter for imaginary part

    double getImaginary() const { return imaginary; }

    // Method to calculate magnitude

    double magnitude() const {

        return sqrt(real \* real + imaginary \* imaginary);

    }

    // Method to calculate angle in radians

    double angle() const {

        return atan2(imaginary, real);

    }

    // Method to calculate complex conjugate

    Complex conjugate() const {

        return Complex(real, -imaginary);

    }

    // Overloaded addition operator

    Complex operator+(const Complex& other) const {

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

    }

    // Overloaded subtraction operator

    Complex operator-(const Complex& other) const {

        return Complex(real - other.real, imaginary - other.imaginary);

    }

    // Overloaded multiplication operator

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

        double resultReal = real \* other.real - imaginary \* other.imaginary;

        double resultImaginary = real \* other.imaginary + imaginary \* other.real;

        return Complex(resultReal, resultImaginary);

    }

    // Overloaded division operator

    Complex operator/(const Complex& other) const {

        double denominator = other.real \* other.real + other.imaginary \* other.imaginary;

        double resultReal = (real \* other.real + imaginary \* other.imaginary) / denominator;

        double resultImaginary = (imaginary \* other.real - real \* other.imaginary) / denominator;

        return Complex(resultReal, resultImaginary);

    }

    // Method to print complex number

    void Print() const {

        std::cout << "(" << real << ", " << imaginary << ")" << std::endl;

    }

};

int main() {

    // Examples of using the Complex class

    Complex a(1, 2);

    Complex b("3, 4");

    Complex sum = a + b;

    Complex difference = a - b;

    Complex product = a \* b;

    Complex quotient = a / b;

    std::cout << "a + b = ";

    sum.Print();

    std::cout << "a - b = ";

    difference.Print();

    std::cout << "a \* b = ";

    product.Print();

    std::cout << "a / b = ";

    quotient.Print();

    return 0;

}

2.

#include <iostream>

#include <vector>

#include <sstream>

#include <stdexcept>

class Matrix {

private:

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

    bool notAMatrix;

public:

    // Constructor

    Matrix(const std::string& matrixString) : notAMatrix(false) {

        std::istringstream iss(matrixString);

        char discard;

        int value;

        std::vector<int> row;

        while (iss >> discard) {

            row.clear();

            while (iss >> value >> discard) {

                row.push\_back(value);

            }

            data.push\_back(row);

        }

        // Check if all rows have the same number of elements

        int numCols = data[0].size();

        for (const auto& row : data) {

            if (row.size() != numCols) {

                notAMatrix = true;

                break;

            }

        }

    }

    // Destructor

    ~Matrix() {

        // No explicit memory deallocation required for std::vector

    }

    // Check if matrix is Not a Matrix

    bool IsNaM() const {

        return notAMatrix;

    }

    // Indexing operator

    std::vector<int>& operator[](size\_t index) {

        if (index >= data.size()) {

            throw std::out\_of\_range("Index out of range");

        }

        return data[index];

    }

    // Matrix addition

    Matrix operator+(const Matrix& other) const {

        if (data.size() != other.data.size() || data[0].size() != other.data[0].size()) {

            throw std::invalid\_argument("Matrix dimensions are not compatible for addition");

        }

        Matrix result = \*this;

        for (size\_t i = 0; i < data.size(); ++i) {

            for (size\_t j = 0; j < data[0].size(); ++j) {

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

            }

        }

        return result;

    }

    // Matrix subtraction

    Matrix operator-(const Matrix& other) const {

        if (data.size() != other.data.size() || data[0].size() != other.data[0].size()) {

            throw std::invalid\_argument("Matrix dimensions are not compatible for subtraction");

        }

        Matrix result = \*this;

        for (size\_t i = 0; i < data.size(); ++i) {

            for (size\_t j = 0; j < data[0].size(); ++j) {

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

            }

        }

        return result;

    }

    // Matrix multiplication

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

        if (data[0].size() != other.data.size()) {

            throw std::invalid\_argument("Matrix dimensions are not compatible for multiplication");

        }

        Matrix result("(");

        for (size\_t i = 0; i < data.size(); ++i) {

            for (size\_t j = 0; j < other.data[0].size(); ++j) {

                int sum = 0;

                for (size\_t k = 0; k < data[0].size(); ++k) {

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

                }

                result[i].push\_back(sum);

            }

            if (i != data.size() - 1) {

                result.data.emplace\_back();

            }

        }

        return result;

    }

    // Print matrix

    void Print() const {

        for (const auto& row : data) {

            for (const auto& value : row) {

                std::cout << value << " ";

            }

            std::cout << std::endl;

        }

    }

};

int main() {

    std::string matrixString1 = "(1,2,3),(4,5,6),(7,8,9)";

    std::string matrixString2 = "(9,8,7),(6,5,4),(3,2,1)";

    Matrix matrix1(matrixString1);

    Matrix matrix2(matrixString2);

    std::cout << "Matrix 1:" << std::endl;

    matrix1.Print();

    std::cout << std::endl;

    std::cout << "Matrix 2:" << std::endl;

    matrix2.Print();

    std::cout << std::endl;

    try {

        Matrix sum = matrix1 + matrix2;

        std::cout << "Sum:" << std::endl;

        sum.Print();

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

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

    }

    return 0;

}