**#Solution-1:**

#include <iostream>

#include <cmath> // For mathematical functions

class Complex {

private:

double real; // Real part

double imaginary; // Imaginary part

public:

// Default constructor

Complex() : real(0.0), imaginary(0.0) {}

// Constructor with two arguments (real and imaginary parts)

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

// Constructor from a string (e.g., "123, 456")

Complex(const std::string& complexStr) {

// Parse the string to extract real and imaginary parts

size\_t commaPos = complexStr.find(',');

if (commaPos != std::string::npos) {

real = std::stod(complexStr.substr(0, commaPos));

imaginary = std::stod(complexStr.substr(commaPos + 1));

}

}

// Calculate the magnitude (absolute value) of the complex number

double magnitude() const {

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

}

// Calculate the angle (in radians) of the complex number

double angle() const {

return std::atan2(imaginary, real);

}

// Calculate the complex conjugate

Complex conjugate() const {

return Complex(real, -imaginary);

}

// Print the complex number

void Print() const {

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

}

// Overload addition operator

Complex operator+(const Complex& other) const {

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

}

// Overload subtraction operator

Complex operator-(const Complex& other) const {

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

}

// Overload multiplication operator

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

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

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

return Complex(newReal, newImaginary);

}

// Overload division operator

Complex operator/(const Complex& other) const {

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

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

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

return Complex(newReal, newImaginary);

}

};

int main() {

Complex c1(1, 2);

Complex c2("3, 4");

Complex sum = c1 + c2;

Complex difference = c1 - c2;

Complex product = c1 \* c2;

Complex quotient = c1 / c2;

std::cout << "Sum: ";

sum.Print();

std::cout << std::endl;

std::cout << "Difference: ";

difference.Print();

std::cout << std::endl;

std::cout << "Product: ";

product.Print();

std::cout << std::endl;

std::cout << "Quotient: ";

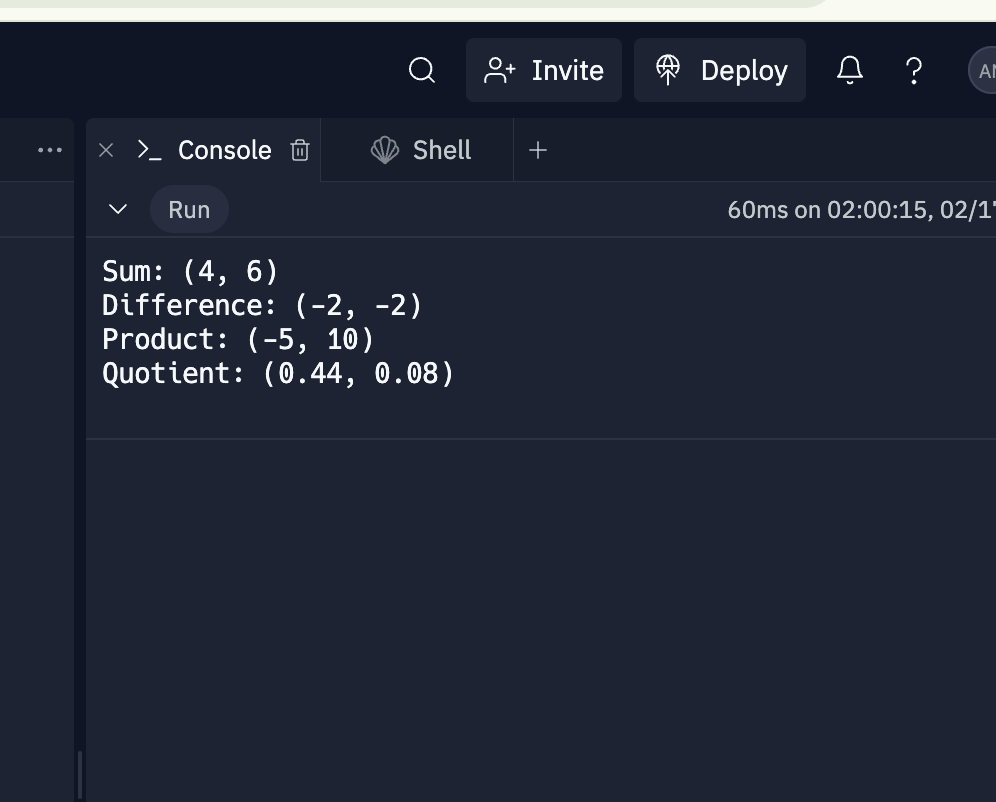
quotient.Print();

std::cout << std::endl;

return 0;

}

Output:



#Solution-2:

#include <iostream>

#include <vector>

#include <string>

#include <sstream>

#include <stdexcept>

class Matrix {

private:

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

bool isNaM;

public:

Matrix(const std::string& input) {

std::stringstream ss(input);

int num;

std::vector<int> row;

while (ss >> num) {

row.push\_back(num);

if (ss.peek() == ',')

ss.ignore();

else if (ss.peek() == ';') {

data.push\_back(row);

row.clear();

ss.ignore();

}

}

data.push\_back(row);

size\_t cols = data.empty() ? 0 : data[0].size();

isNaM = false;

for (const auto& r : data) {

if (r.size() != cols) {

isNaM = true;

break;

}

}

}

bool IsNaM() const {

return isNaM;

}

int& operator()(size\_t i, size\_t j) {

if (i >= data.size() || j >= data[0].size())

throw std::out\_of\_range("Matrix index out of bounds.");

return data[i][j];

}

const int& operator()(size\_t i, size\_t j) const {

if (i >= data.size() || j >= data[0].size())

throw std::out\_of\_range("Matrix index out of bounds.");

return data[i][j];

}

};

int main() {

try {

Matrix mat("(1,2,3;4,5,6;7,8,9)");

if (mat.IsNaM()) {

std::cout << "Matrix is not valid.\n";

} else {

std::cout << "Matrix is valid.\n";

std::cout << "The element at (0, 0) is " << mat(0, 0) << "\n";

}

} catch (const std::exception& e) {

std::cerr << "Error: " << e.what() << '\n';

return 1;

}

return 0;

}

**Output:**

