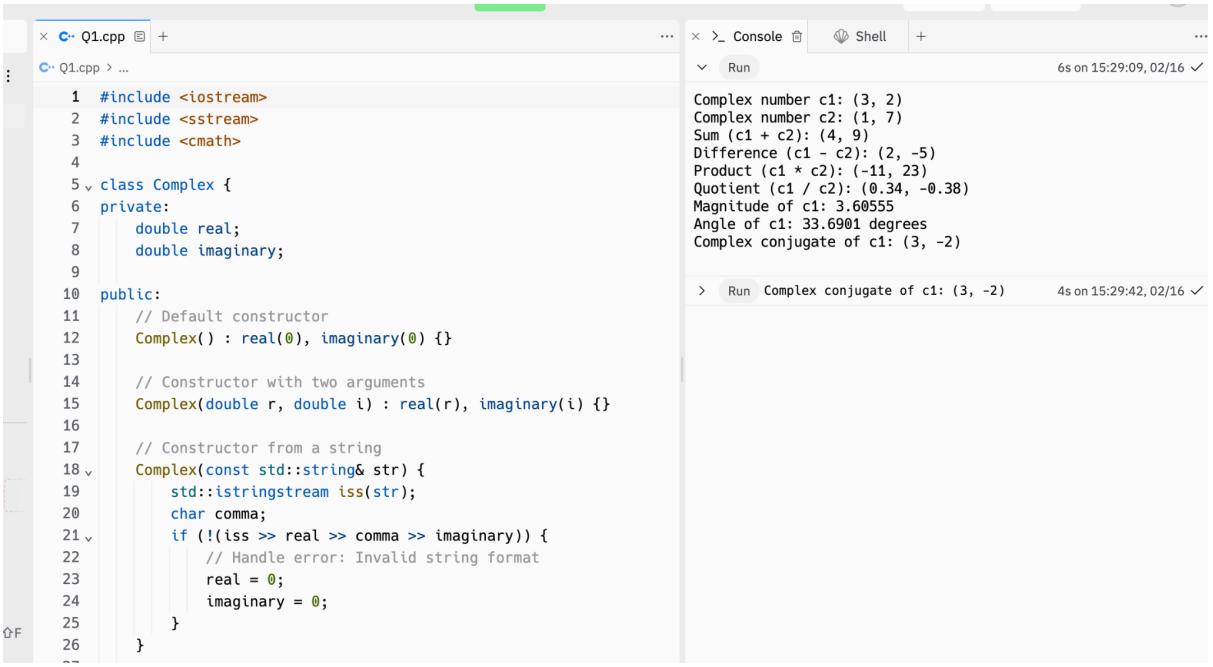
Q.No.1)

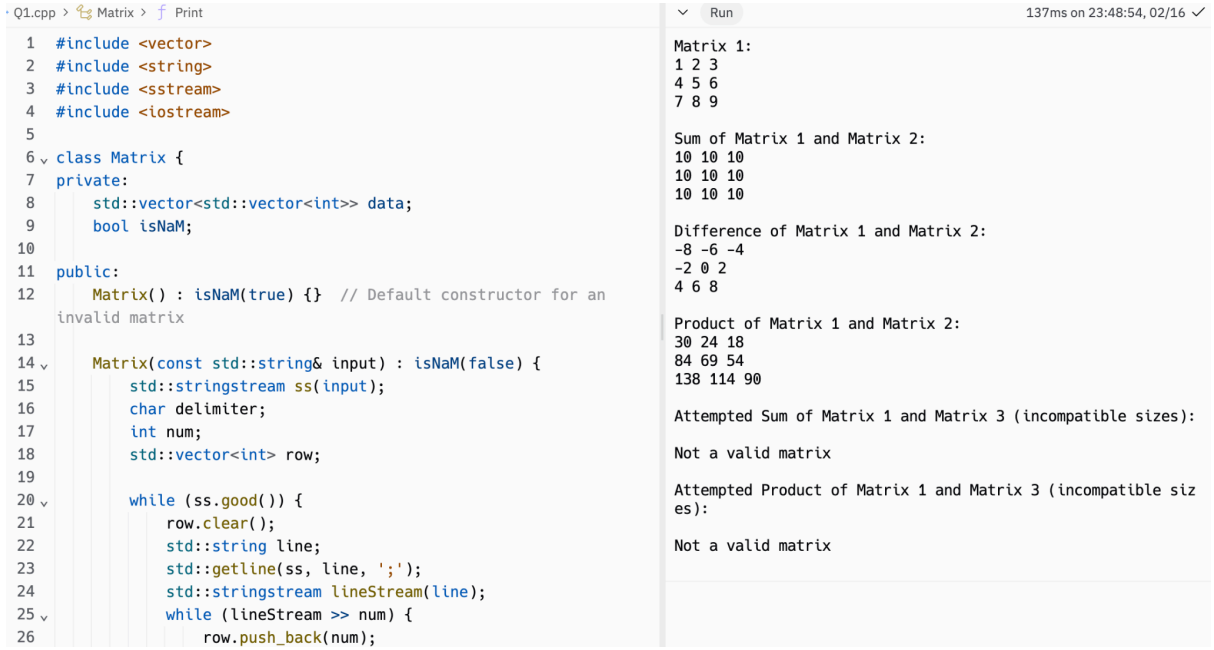



```
#include <iostream>
#include <sstream>
#include <cmath>

class Complex {
private:
    double real;
    double imaginary;

public:
    // Default constructor
    Complex() : real(0), imaginary(0) {}

    // Constructor with two arguments
    Complex(double r, double i) : real(r), imaginary(i) {}

    // Constructor from a string
    Complex(const std::string& str) {
        std::istringstream iss(str);
        char comma;
        if (!(iss >> real >> comma >> imaginary)) {
            // Handle error: Invalid string format
            real = 0;
            imaginary = 0;
        }
    }
```

```
    // Addition
    Complex operator+(const Complex& other) const {
        return Complex(real + other.real, imaginary + other.imaginary);
    }

    // Subtraction
    Complex operator-(const Complex& other) const {
        return Complex(real - other.real, imaginary - other.imaginary);
    }

    // Multiplication
    Complex operator*(const Complex& other) const {
        double r = real * other.real - imaginary * other.imaginary;
        double i = real * other.imaginary + imaginary * other.real;
        return Complex(r, i);
    }

    // Division
    Complex operator/(const Complex& other) const {
        double denominator = other.real * other.real + other.imaginary * other.imaginary;
        double r = (real * other.real + imaginary * other.imaginary) / denominator;
        double i = (imaginary * other.real - real * other.imaginary) / denominator;
        return Complex(r, i);
    }

    // Magnitude
    double magnitude() const {
        return sqrt(real * real + imaginary * imaginary);
    }

    // Angle in degrees
    double angle() const {
        return atan2(imaginary, real) * (180 / M_PI);
    }

    // Complex conjugate
    Complex conjugate() const {
        return Complex(real, -imaginary);
    }

    // Print
    void Print() const {
        std::cout << "(" << real << ", " << imaginary << ")" << std::endl;
    }
};

int main() {
```

```
    Complex c1(3, 2);
    Complex c2(1, 7);

    std::cout << "Complex number c1: ";
    c1.Print();

    std::cout << "Complex number c2: ";
    c2.Print();

    // Addition
    Complex sum = c1 + c2;
    std::cout << "Sum (c1 + c2): ";
    sum.Print();

    // Subtraction
    Complex difference = c1 - c2;
    std::cout << "Difference (c1 - c2): ";
    difference.Print();

    // Multiplication
    Complex product = c1 * c2;
    std::cout << "Product (c1 * c2): ";
    product.Print();

    // Division
    Complex quotient = c1 / c2;
    std::cout << "Quotient (c1 / c2): ";
    quotient.Print();

    // Magnitude of c1
    std::cout << "Magnitude of c1: " << c1.magnitude() << std::endl;

    // Angle of c1 in degrees
    std::cout << "Angle of c1: " << c1.angle() << " degrees" << std::endl;

    // Complex conjugate of c1
    Complex conjugate = c1.conjugate();
    std::cout << "Complex conjugate of c1: ";
    conjugate.Print();


    return 0;
}
```

Q.No.2)




```cpp
#include <vector>
#include <string>
#include <sstream>
#include <iostream>

class Matrix {
private:
    std::vector<std::vector<int>> data;
    bool isNaM;

public:
    Matrix() : isNaM(true) {}  // Default constructor for an invalid matrix

    Matrix(const std::string& input) : isNaM(false) {
        std::stringstream ss(input);
        char delimiter;
        int num;
        std::vector<int> row;

        while (ss.good()) {
            row.clear();
            std::string line;
            std::getline(ss, line, ';');
            std::stringstream lineStream(line);
            while (lineStream >> num) {
                row.push_back(num);
                lineStream >> delimiter;
            }
            if (!data.empty() && row.size() != data[0].size()) {
                isNaM = true;
```

```
                data.clear();
                break;
            }
            data.push_back(row);
        }
    }

    bool IsNaM() const {
        return isNaM || data.empty();
    }

    Matrix operator+(const Matrix& other) const {
        if (isNaM || other.isNaM || data.size() != other.data.size() || data[0].size() !=
other.data[0].size()) {
            return Matrix(); // Return an invalid matrix
        }

        Matrix result;
        result.data.resize(data.size(), std::vector<int>(data[0].size(), 0));
        result.isNaM = false;
        for (size_t i = 0; i < data.size(); i++) {
            for (size_t j = 0; j < data[0].size(); j++) {
                result.data[i][j] = data[i][j] + other.data[i][j];
            }
        }
        return result;
    }

    Matrix operator-(const Matrix& other) const {
        if (isNaM || other.isNaM || data.size() != other.data.size() || data[0].size() !=
other.data[0].size()) {
            return Matrix(); // Return an invalid matrix
        }

        Matrix result;
        result.data.resize(data.size(), std::vector<int>(data[0].size(), 0));
        result.isNaM = false;
        for (size_t i = 0; i < data.size(); i++) {
            for (size_t j = 0; j < data[0].size(); j++) {
                result.data[i][j] = data[i][j] - other.data[i][j];
            }
        }
        return result;
    }

    Matrix operator*(const Matrix& other) const {
        if (isNaM || other.isNaM || data[0].size() != other.data.size()) {
            return Matrix(); // Return an invalid matrix
```

```
        }

        Matrix result;
        result.isNaM = false;
        result.data.resize(data.size(), std::vector<int>(other.data[0].size(), 0));
        for (size_t i = 0; i < data.size(); i++) {
            for (size_t j = 0; j < other.data[0].size(); j++) {
                for (size_t k = 0; k < data[0].size(); k++) {
                    result.data[i][j] += data[i][k] * other.data[k][j];
                }
            }
        }
        return result;
    }

    void Print() const {
        if (IsNaM()) {
            std::cout << "\nNot a valid matrix due to incompatible sizes or internal error.\n";
            return;
        }
        for (const auto& row : data) {
            for (const auto& elem : row) {
                std::cout << elem << " ";
            }
            std::cout << "\n";
        }
    }
};

int main() {
    Matrix m1("1,2,3;4,5,6;7,8,9");
    Matrix m2("9,8,7;6,5,4;3,2,1");
    Matrix m3("1,2;3,4"); // Incompatible for addition/multiplication with m1 and m2

    Matrix sum = m1 + m2;
    Matrix difference = m1 - m2;
    Matrix product = m1 * m2;

    Matrix invalidSum = m1 + m3; // Should result in an invalid matrix
    Matrix invalidProduct = m1 * m3; // Also should result in an invalid matrix

    std::cout << "Matrix 1:\n";
    m1.Print();

    std::cout << "\nSum of Matrix 1 and Matrix 2:\n";
    sum.Print();

    std::cout << "\nDifference of Matrix 1 and Matrix 2:\n";
```

```
    difference.Print();

    std::cout << "\nProduct of Matrix 1 and Matrix 2:\n";
    product.Print();

    std::cout << "\nAttempted Sum of Matrix 1 and Matrix 3 (incompatible sizes):\n";
    invalidSum.Print();

    std::cout << "\nAttempted Product of Matrix 1 and Matrix 3 (incompatible sizes):\n";
    invalidProduct.Print();

return 0;

}
```