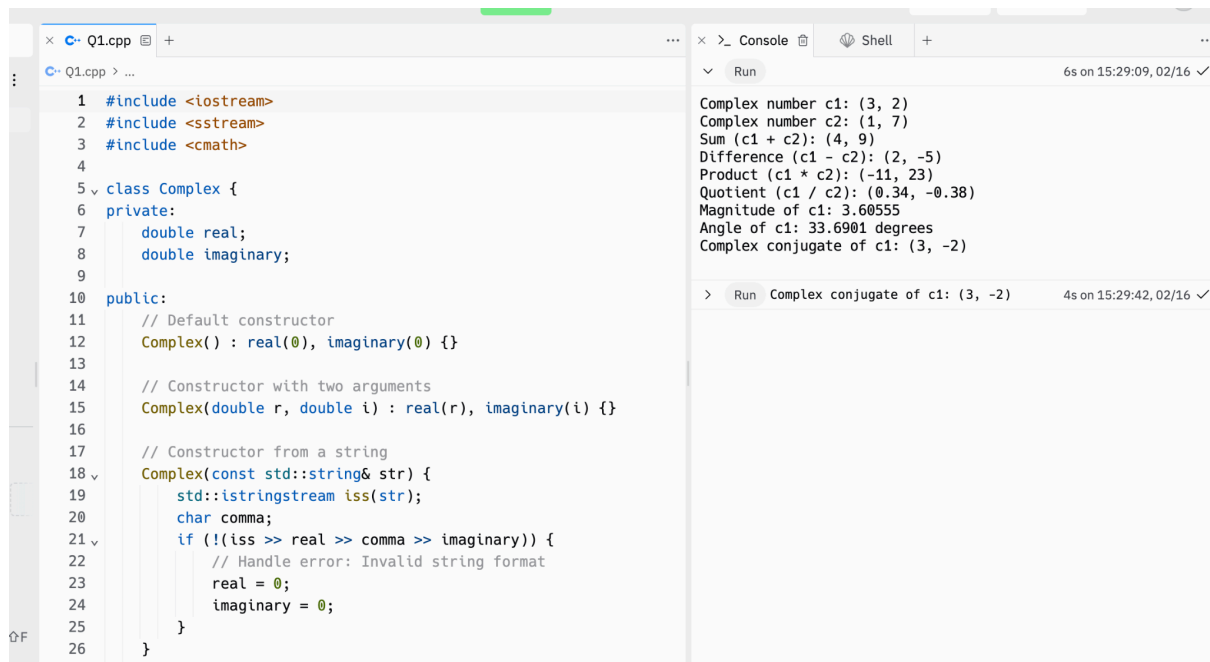


Q.No.1)



```
1 #include <iostream>
2 #include <sstream>
3 #include <cmath>
4
5 class Complex {
6 private:
7     double real;
8     double imaginary;
9
10 public:
11     // Default constructor
12     Complex() : real(0), imaginary(0) {}
13
14     // Constructor with two arguments
15     Complex(double r, double i) : real(r), imaginary(i) {}
16
17     // Constructor from a string
18     Complex(const std::string& str) {
19         std::istringstream iss(str);
20         char comma;
21         if (!(iss >> real >> comma >> imaginary)) {
22             // Handle error: Invalid string format
23             real = 0;
24             imaginary = 0;
25         }
26     }
27 }
```

Complex number c1: (3, 2)
Complex number c2: (1, 7)
Sum (c1 + c2): (4, 9)
Difference (c1 - c2): (2, -5)
Product (c1 * c2): (-11, 23)
Quotient (c1 / c2): (0.34, -0.38)
Magnitude of c1: 3.60555
Angle of c1: 33.6901 degrees
Complex conjugate of c1: (3, -2)

Run Complex conjugate of c1: (3, -2) 4s on 15:29:42, 02/16 ✓

```
#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)

```
Q1.cpp > Matrix > f Print 137ms on 23:48:54, 02/16 ✓
1 #include <vector>
2 #include <string>
3 #include <sstream>
4 #include <iostream>
5
6 class Matrix {
7 private:
8     std::vector<std::vector<int>> data;
9     bool isNaM;
10
11 public:
12     Matrix() : isNaM(true) {} // Default constructor for an
    invalid matrix
13
14     Matrix(const std::string& input) : isNaM(false) {
15         std::stringstream ss(input);
16         char delimiter;
17         int num;
18         std::vector<int> row;
19
20         while (ss.good()) {
21             row.clear();
22             std::string line;
23             std::getline(ss, line, ';');
24             std::stringstream lineStream(line);
25             while (lineStream >> num) {
26                 row.push_back(num);
```

```
Matrix 1:
1 2 3
4 5 6
7 8 9

Sum of Matrix 1 and Matrix 2:
10 10 10
10 10 10
10 10 10

Difference of Matrix 1 and Matrix 2:
-8 -6 -4
-2 0 2
4 6 8

Product of Matrix 1 and Matrix 2:
30 24 18
84 69 54
138 114 90

Attempted Sum of Matrix 1 and Matrix 3 (incompatible sizes):
Not a valid matrix

Attempted Product of Matrix 1 and Matrix 3 (incompatible sizes):
Not a valid matrix
```

```
#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;

}
```