Operator Overloading

1. **Complex Number Arithmetic using Operator Overloading**

Problem Statement: Implement a class `Complex` that represents complex numbers. Overload the following operators:

- **Addition (`+`)**: To add two complex numbers.
- **Subtraction (`-`)**: To subtract two complex numbers.
- **Multiplication (`*`)**: To multiply two complex numbers.
- **Division (`/`)**: To divide two complex numbers.

Provide a `print()` method to display the result of complex number operations in the format `(real + imaginary i)`.

Sample Input:

```
Complex c1(4, 5);  // 4 + 5i
Complex c2(2, 3);  // 2 + 3i
Complex c3 = c1 + c2;
c3.print();  // Expected Output: (6 + 8i)
```

2. **Matrix Addition and Scalar Multiplication using Operator Overloading**

Problem Statement: Implement a class `Matrix` for 2x2 matrices. Overload the following operators:

- **Addition (`+`)**: To add two matrices.
- **Scalar multiplication (`*`)**: To multiply a matrix by a scalar value.

Also, implement a method `display()` to print the matrix in a readable format.

Sample Input:

```
Matrix m1(1, 2, 3, 4);  // Matrix: [1, 2], [3, 4]
Matrix m2(5, 6, 7, 8);  // Matrix: [5, 6], [7, 8]
Matrix m3 = m1 + m2;
m3.display();  // Expected Output: [6, 8], [10, 12]

Matrix m4 = m1 * 2;
m4.display();  // Expected Output: [2, 4], [6, 8]
```

3. **String Concatenation using Operator Overloading**

Problem Statement: Implement a class `MyString` that behaves like a string. Overload the `+` operator to concatenate two strings and the `[]` operator to access characters at a specific index.

Sample Input:

```
MyString str1("Hello");
MyString str2("World");
MyString str3 = str1 + str2;
std::cout << str3;  // Expected Output: HelloWorld

std::cout << str3[0];  // Expected Output: H
```

4. **Rational Number Arithmetic using Operator Overloading**

Problem Statement: Implement a class `Rational` to represent rational numbers. Overload the following operators:

- **Addition (`+`)**: To add two rational numbers.
- **Subtraction (`-`)**: To subtract two rational numbers.
- **Multiplication (`*`)**: To multiply two rational numbers.
- **Division (`/`)**: To divide two rational numbers.
- **Equality (`==`)**: To check if two rational numbers are equal.

Use the `gcd` function to reduce the fraction to its simplest form.

Sample Input:

```
Rational r1(2, 3);  // 2/3
Rational r2(3, 4);  // 3/4
Rational r3 = r1 + r2;
r3.display();  // Expected Output: 17/12

if (r1 == r2) {
    std::cout << "Equal" << std::endl;
} else {
    std::cout << "Not Equal" << std::endl;  // Expected Output: Not Equal
}
```

5. **Fraction Class with Comparison and Arithmetic Operators**

Problem Statement: Implement a class `Fraction` to represent fractions. Overload the following operators:

- **Addition (`+`)**: To add two fractions.
- **Subtraction (`-`)**: To subtract two fractions.
- **Multiplication (`*`)**: To multiply two fractions.
- **Division (`/`)**: To divide two fractions.
- **Comparison (`<`, `>`, `==`)**: To compare if one fraction is smaller, greater, or equal to another.

Ensure that the fractions are always stored in the lowest terms by reducing them after every operation using the greatest common divisor (GCD).

Sample Input:

```
Fraction f1(2, 3);  // 2/3
Fraction f2(3, 4);  // 3/4
Fraction f3 = f1 + f2;
f3.display();  // Expected Output: 17/12

if (f1 < f2) {
    std::cout << "f1 is smaller than f2" << std::endl;  // Expected Output: f1 is
smaller than f2
}
```
