# Problem Sheet 2

## LM Advanced Mathematical Finance Module

### Part B C++ Spring Term 2021

| PROBLEM SHEET 2 | SPRING TERM, PROGRAMMING IN C++ LECTURE 2 |
|---|---|
| Lecturer | Dr Simon Hartley s.hartley@bham.ac.uk |
| Of | School of Mathematics, College of Engineering and Physical Sciences, University of Birmingham |
| Due | Tuesday 20th April 2021, 23:59 |
| Weighting | 20% of Part B |

### INSTRUCTIONS FOR THE SUBMISSION

**Submit your code on Canvas as zip:**

Submit as a single zip file containing all code for the problem.

The source files should compile without errors or warnings on Visual Studio 2019 as installed on the clusters, and the resulting executables should run without problems. You do not have to submit the executable.

**If you use another compiler other than Visual Studio, you should take your final code and compile it on Visual Studio on the remote clusters.**

Unfortunately, different compilers follow slightly different rules, and even a small difference may cause the code to fail on Visual Studio. The source code should also be well formatted. This means that opening and closing brackets should be aligned in a readable way, and we will discuss such styles briefly in the lectures. Finally, the programs you write should run in a self-explanatory way. If you are asked that the program provides some input, it should first have some output on the command line telling the user what kind of input is required.

**For instance,** if you are supposed to read in the price of a interest rate, do the following:

```
double interest_rate;
std::cout << “ Please enter the interest rate: “ << std::endl;
std::cin >> interest_rate;
```

This way, a user of the program who does not know about the code knows what numbers one has to enter.

### PLAGIARISM

Obviously, you are not allowed to plagiarise. This means that you are not allowed to directly copy the code from any online sources or any of your fellow students. Direct group work is also not allowed. While you are allowed (and even encouraged) to discuss the contents of this module (including problem sheets) with your fellow students, every one of you needs to write your own code.

### PROBLEM 1 Create a C++ class to implement rational numbers

When you finish this problem, you should have:

- Implemented a C++ class.
- Implemented a set of functions that use the class.
- Implemented a program that employs a C++ class and utility functions.
- Implemented a program that tests for all your operators and functions.
- Implemented a program that tests your utillity operators and functions.

When this assignment is complete, your code should produce three executable programs: RationalCalculator, RationalTest and RationalPrintTest. So you should have five source code files (.cpp), and two header files (.h).

## Description

Rationals represent the ratio of two integers a numerator and a denominator. In canonical/reduced form, the greatest common divisor (GCD) between the numerator and the denominator should be 1. For this problem you will be implementing and using a Rational Number class. This assignment will consist of several parts:

- Your implementation of the cRational class.
- A test, cRationalTest, for the cRational class
- Your implementation of a output library of support functions.
- A test, RationalPrintTest, for your output library
- Your implementation of a simple calculator

For the first part, implement a cRational class.

Your class should be defined by **cRational.h** and **cRational.cpp**.

```
    // Initialize the rational The default value should be 0/1
    // Default Constructor
    cRational();
    // Complete Constructor
    cRational(long long n, long long d);

    // Copy Constructor
    cRational(const cRational& b);
    // Destructor
    ~cRational();

    // Set the numerator to the given value. The denominator should be set to be 1.
    void Set(long long numerator);

     // Set both the numerator and denominator. Reduce to canonical form the rational if necessa
    void Set(long long numerator, long long denominator);

    void SetNumerator(long long numerator);
    void SetDenominator(long long denominator);

    // Return the numerator.
    long long Numerator(void)const;

    // Return the denominator.
    long long Denominator(void)const;

    // Return the floating point value of the rational.
    double RealValue(void) const;

    // Add two rationals producing a new rational. The new rational should be canonical form.
    cRational operator +(const cRational& b) const;

    // Add Function producing a new rational.
    cRational Add(cRational& r1, cRational& r2)const;

    // Subtract two rationals producing a new rationals. The new rationals should be canonical f
    cRational operator -(const cRational& b) const;

    // Multiply two rationals producing a new rationals. The new rationals should be canonical f
    cRational operator *(const cRational& b) const;
    friend cRational operator*(const cRational& r1, long long value);
    friend cRational operator*(long long value, const cRational& f1);
    cRational& operator*=(const cRational& rhs);
    // Divide two rationals producing a new rational. The new rational should be canonical form.
    cRational operator /(const cRational& b) const;

    // Comparison of two rationals
    friend bool operator < (const cRational& lhs, const cRational& rhs);

    // once operator< is provided, the other relational operators are implemented in terms of op
    friend bool operator > (const cRational& lhs, const cRational& rhs);

    // Implement the operator<
    friend bool operator <=(const cRational& lhs, const cRational& rhs);

    // Implement the operator<
    friend bool operator >=(const cRational& lhs, const cRational& rhs);

    // Implement the operator ==
    friend bool operator ==(const cRational& lhs, const cRational& rhs);

    // the inequality operator is typically implemented in terms of operator==
    friend bool operator !=(const cRational& lhs, const cRational& rhs);

    // access  the numerator [0] or denominator [1]
    long long& operator[](std::size_t idx);

    // access  the numerator [0] or denominator [1]
    const long long& operator[](std::size_t idx) const;

    // produce a new rational which is the opposite of this
    cRational opposite(void)const;

    // produce a new rational which is the reciprocal of this
    cRational reciprocal(void)const;

    // produce a new rational which is Exponentiation to integer power n
    cRational exp(const int n)const;

    // Compute the Greatest Comon Denominator of two integers. See notes below.
    long long GCD(long long a, long long b) const;

    // Reduce the rational to its canonical form
    void canonicalform(void);
    // We'll make gcd static so that it can be part of class rational without
    // requiring an object of type cRational
    static long long Greatest_Comon_Denominator(long long a, long long b);
    friend std::ostream& operator<<(std::ostream& os, const cRational& a);
    friend std::istream& operator>>(std::istream& is, cRational& obj);
```

### Greatest Common Denominator

The algorithm for this function is given as

```
function gcd(a, b)
    while b ≠ 0
        t := b
        b := a mod b
        a := t
    return a
```

Implement a series tests for your class called from a main function which is defined in RationalTest.cpp.

Note: **you should probably do this as you are developing the classes**

Next implement a pair of utility functions in FunctionUtils.cpp and FunctionUtils.h.

Using FunctionUtils.h implement the two routines PrintRational and PrintReducedRational. You should provide methods which use the getters and an alternative which uses friend functions named appropriately.

**PrintcRational** prints the given input as an improper function. The following rules apply

- If the numerator is 0, print a 0.
- If the denominator is 1, just print the numerator.
- Otherwise print numerator/denominator
- Do not print an endl.

**PrintMixedRational** uses the same rules with the exception that if the numerator is greater than the denominator, the rational is printed as a mixedrRational. i.e. 4/3 should be printed as 1 1/3.

- Print a space between the integer and the rational.
- Print the negative sign with the integer.

Implement a test for your utility functions called from a main function which is defined in RationalPrintTest.cpp

Implement a command line calculator that uses Rationals. in a file called RationalCalculator.cpp which contains a main function.

This calculator should prompt with “>”, read the line that consist of an expression in the from Rational operation Rational, and prints the result.

The operations supported should be +, -, *, /. All components will be separated by a space and the expression will be terminated by an newline. You may assume that there will be no errors in the input and that there are no spaces in the rationals, including the sign of the rational.

If the first rational is “quit”, the program should exit.

An example run should look like this

```
> 1/4 + 3/4
1
> 1/4 * 1/4
1/16
> 4/6 + 4/6
1 1/3
> 1/4 + -1/4
0
> -1/2 + -1/2
-1
> quit
```

## Notes

- At no time should the denominator be allowed to become 0.
- If this would occur due to a call to Set, do not change the value of the rational.
- If this occurs due to an operation (0/1 / 7/2) we really should say we have an error.
- Rationals must always be in canonical form.
- If a rational is negative, the negative sign should be associated with the numerator.

```
    - 1/9 should be stored as -1/9
     -5/-9 should be stored as 5/9
      3/-7 should be stored as -3/7
```

- When computing GCD make sure both numbers are positive.

## Bonus for Advanced

- Use a makefile project (an option in visual studio) submit a makefile which produces the executables
- generate and handle the error for the denominator being zero or for array operator indexing