C++ Exercise 4: Matrix Operations

Anotācija

This exercise covers polymorphism, function overloading as well as operator overloading, raising domain-specific exceptions or error conditions.

How to submit: Check your code into your GitHub repository, the default `master` branch, tag it as `ex04submit` (all lowercase, no dashes).
Grading: This exercise is worth 30‰(or 3%) of the total grade.

Develop a software that receives an input file containing two rectangular matrices (in our math expressions we denote them by A and B respectively). Every matrix first displays its type (`MZ`, `MQ` or `MR`), its dimensions (positive integers: row number m, column number n), followed by exactly $m \cdot n$ numbers of the respective type. Both matrices are of the same type.
After that it receives an operation name (`ADD`, `SUB` or `MUL`) and writes the resulting matrix to the standard output. If matrix sizes do not allow the operation to be performed, throw `std::out_of_range`.
Valid matrix operations: Assume that there is a matrix A of size $m_1 \times n_1$ and a matrix B of size $m_2 \times n_2$:

(A) $C = A + B$ (operation `ADD`) is doable iff $m_1 = m_2$ and $n_1 = n_2$ (both sizes match). The elements (c_{ij}) of C are defined as $c_{ij} = a_{ij} + b_{ij}$.

(B) $C = A - B$ (operation `SUB`) is doable iff $m_1 = m_2$ and $n_1 = n_2$ (both sizes match). The elements (c_{ij}) of C are defined as $c_{ij} = a_{ij} - b_{ij}$.

(C) $C = A \cdot B$ (operation `MUL`) is doable iff $n_1 = m_2$ (the number of colums in the matrix to the left equals the number of rows in the matrix to the right). The elements (c_{ij}) of C are defined as

$$c_{ij} = \sum_{k=1}^{n_1} a_{ik} \cdot b_{kj},$$

kur $i \in \{1, \ldots, m_1\}$ un $j \in \{1, \ldots, n_2\}$.

Input
The first line contains abbreviation `MZ`, `MQ` and `MR` followed by number of rows `m` and the number of columns `n` (two positive integers). After that there are $m \cdot n$ numbers (integers, rationals written with a forward slash `p/q` or reals – double type numbers).

Output
The output is either the result matrix (of the same type as the two input matrices). It shows type (MZ, MQ or MR) on the first line, also the count of rows and columns. After that it shows all the matrix elements - separated by spaces, line by line. If there was an exception, print just the exception name `out_of_range`.

Limitations

- Any matrix in the input file has the number of rows and the number of columns between 1 and 256 (thus the largest matrix can be 256×256).

- Integer arithmetic when adding, subtracting or computing sums like $a_{i1}b_{1j} + \ldots$ should not overflow the size of `int` register. (Namely, you should not worry about the order in which you add numbers themselves or their products.)
- The same assumption can be made about the rationals (class `Ratio`) as well as reals: You can add them in any order you want. The input data will not deliberately cause large rounding errors as in this example:

$$1000000000 + 0.00001 - 1000000000.$$

- All the rational numbers (both input and output) are in the reduced form ($\gcd(p, q) = 1$), denominators are always positive. For example, the rational number -0.5 is always input and output as $-1/2$ (and never as $-6/3$ or $1/-2$).
- Rational numbers that happen to be integers still have denominator 1 explicitly written. For example, the rational number 17 in `MQ` matrices is always written as `17/1`.
- All matrices in the input will be presented correctly, but sometimes the operations may be impossible.

Notes on Strassen's Algorithm

It could be tempting to consider Strassen's Algorithm for multiplying matrices that are sufficiently large (the number of arithmetic operations would be about $O(n^{2.8074})$ instead of $O(n^3)$). On the other hand, it is likely require very large matrices (at least a few thousand by a few thousand) to see any savings when compared with the school algorithm for matrix multiplication. See `https://bit.ly/3jblOWX` for more discussion on this.

We will have other examples in this course where algorithm can be designed to be more efficient. You could try to implement Strassen's algorithm, if the current task seems too straightforward.

Sample input **test01in.txt**:

```
MZ 3 3
2 -6 8
10 6 -1
-4 -5 0
MZ 3 4
2 -6 -10 -1
6 -5 10 0
9 4 5 -4
MUL
```

Expected output **test01expected.txt**:

```
MZ 3 4
40 50 -40 -34
47 -94 -45 -6
-38 49 -10 4
```

Sample input **test02in.txt**:

```
MQ 2 2
0/1 6/5
-5/9 -1/1
MQ 2 2
-7/4 3/10
5/9 1/8
ADD
```

Expected output **test02expected.txt**:

```
MQ 2 2
-7/4 3/2
0 -7/8
```

Sample input **test03in.txt**:

```
MR 2 3
1 0.3 0
0 0.7 1
MR 2 2
1.0 0
0 1.0
MUL
```

Expected output **test03expected.txt**:

```
out_of_range
```

Implementation Details

1. Implement files `Ratio.h` (optionally also `Ratio.cpp`), `Matrix.h` (optionally also `Matrix.cpp`) and also `MatricMain.cpp`. (You may find that header-only implementation is the most easy to build on most C++ compilers, but using CPP files isfine.)

2. It is recommended to implement the following UML diagram using operator overloading (see Figure 1).
   Note: Just as in previous tasks, your main concern should be passing the Standard Input/Output testcases, since your grade for EX04 only depends on these. Still, we strongly recommend to follow the UML diagram since the following exercise (EX05) will assume that the UML diagram is implemented; it will build on top of these overloaded operators and interfaces.

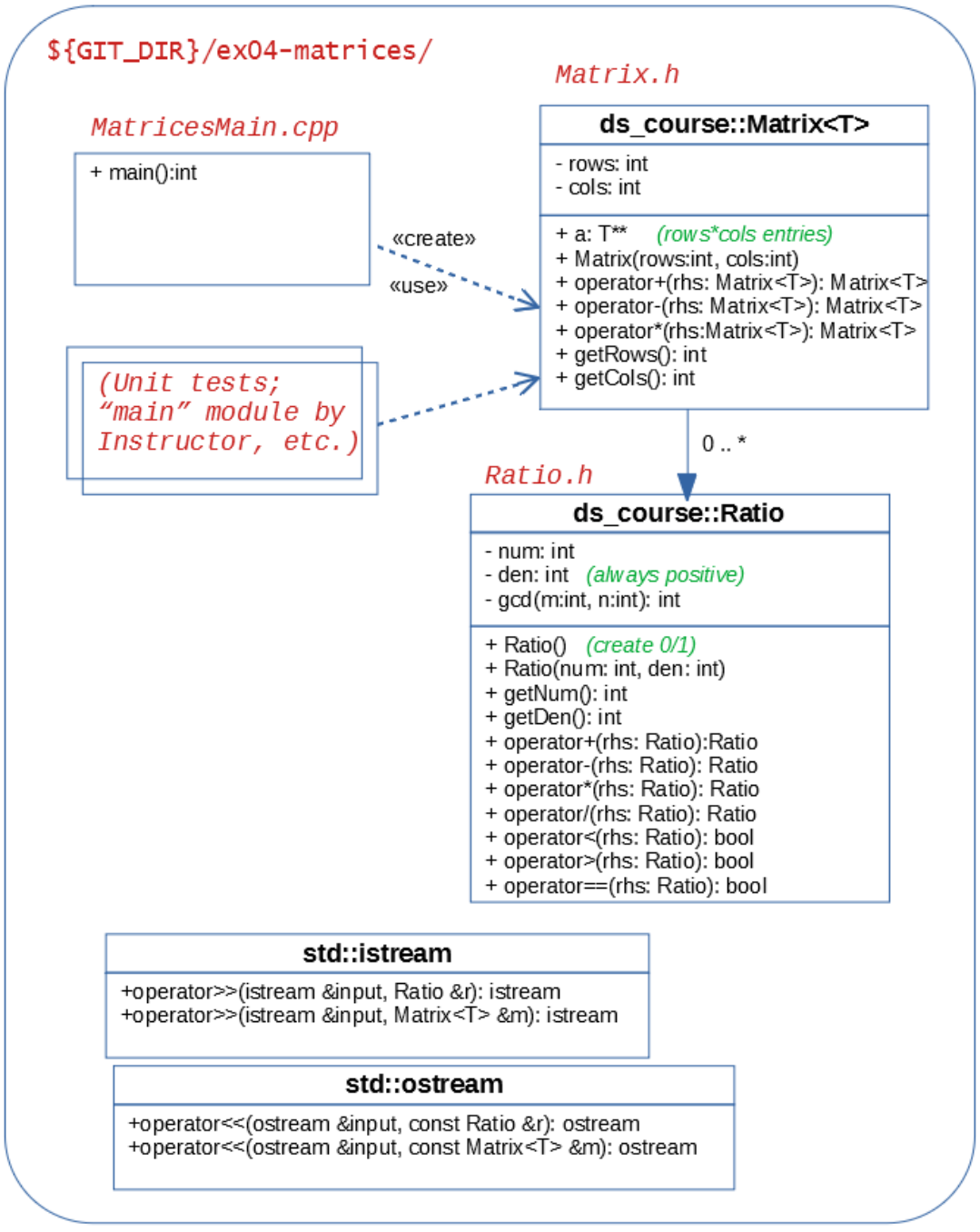

Figure 1: Classes and functions suggested for EX04.