NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C10**

Matrix

The goal of this task is to design and implement a class enabling the representation of matrices containing elements of any numerical type, and subsequently implement several selected basic operations over them.

In particular, we implement the matrix class as a templated class, with parameters in the form of `template<typename element, size_t height, size_t width>`, where `element` is an element data type (e.g., `int`, but also an appropriate user-defined class implementing the necessary behavior), `height` is the matrix height (number of rows), and `width` its width (number of columns).

The internal storage for matrix elements is expected to be implemented via a private data item using the `std::array` container in such a way, that we unfold the entire matrix into one flat linear structure. In other words, this array must not contain other nested arrays, but directly the individual elements. Specifically, let us assume an index arithmetic where an element at logical coordinates `[row][column]` will be placed at position `[row * width + column]` within this internal array.

For the matrix construction, we offer a single parameterized constructor in the form of `Matrix(const element& value = 0)`, where all positions in the created matrix will be filled in with a specified sample element `value`. For basic work with our matrix, we offer the following member functions:

- `const element& get(size_t row, size_t column) const`: returns the current value of an element at the indicated logical coordinates, i.e., at row `row` and column `column`
- `void set(size_t row, size_t column, const element& value)`: sets the new value of an element at the specified logical coordinates
- `void print(std::ostream& stream = std::cout) const`: prints the content of the entire matrix into a given output stream in the following form: first, we place a list of all individual matrix rows into an outer pair of square brackets and separate these rows by commas and spaces, we then enclose each of them again in an inner pair of square brackets and separate its individual elements with commas and spaces, too; for completeness, let us provide a particular example: `[[1, 2], [3, 4], [5, 6]]`; we do not add any line breaks at the end; elements themselves will be printed via their operator `<<`

It is generally assumed that we will request access only to elements on valid coordinates, i.e., those not exceeding the actual dimensions of the matrix. Otherwise, the behavior of the respective functions will be undefined. As for the printing of matrices, to increase comfort, we also add an appropriate global function for the `<<` operator for printing into a stream.

Next, we implement the following two operators for pre-incrementation and post-incrementation, in the form of member functions within our matrix. In both cases, by incrementing, we mean that we increase the current value of each individual matrix element by 1. More precisely, that we will call the respective pre-incrementation or post-incrementation operation over each element as such. Let us not forget that we do not have full control over the elements of our matrices in terms of their type, and so we have to preserve the intended semantics (however unusual and weird it would be, incrementation and addition of 1 could have different meanings).

- `Matrix& operator++()`: we perform matrix pre-incrementation, i.e., return a reference to the existing matrix with the new element values already after their incrementation
- `Matrix operator++(int)`: we perform matrix post-incrementation, i.e., we return a newly created copy of the matrix with the original values before the elements of the existing matrix were actually incremented; value of the passed parameter is ignored

Assume now we wanted the users of our matrices to be provided with an interface by which they could use the square bracket operator to access specific matrix elements based on the actual positions of such elements in our internal storage. E.g., `matrix[4]` in a matrix with 3 rows and 2 columns would correspond to accessing an element at logical coordinates `[2][0]`, that is, an element in the third row and first column.

In this way, we would in fact only mediate the exact behavior of the `[]` operator, which the internal array container already offers. For this purpose, it would be enough if we offered methods in the form of `element& operator[](size_t position)` and `const element& operator[](size_t position) const` in our matrix class. Two separate variants are needed to allow the use of such an operator over both modifiable and non-modifiable matrix instances.

Obviously, implementing such a pair of operators would be straightforward, but they would certainly not be user-friendly. We will, therefore, not implement them, and instead, we will introduce square bracket operators that will allow us to access specific elements based on our logical coordinates.

In other words, we want to have the possibility of two-level indexing, first according to a specific row, then according to a specific column. E.g., `matrix[2][0]`. In order for something like this to be possible, it will first be necessary to propose an auxiliary class, through which we will first remember the request for accessing a given selected row and only then execute the request for accessing a specific column as well. In other words, the `[]` operator on the first level (as methods on the matrix class itself) first returns an instance of our auxiliary class, only then the `[]` operator on the second level (as methods on the request class) makes the required element as such available.

To learn further new techniques, we will implement the aforementioned auxiliary request class as an inner private class of the matrix class. In terms of data items, it will contain a constant reference to the matrix (`const Matrix&`) and the requested row number. We deliberately make the corresponding parameterized constructor private and make it available to the matrix class using the `friend` construct. To remove the constancy protection when accessing a specific element, we will use the `const_cast` retyping mechanism.

Finally, we implement the following operations on matrices, this time using global functions in all cases:

-
  ```
  Matrix<element, height, width> operator+(
    const Matrix<element, height, width>& matrix,
    const element& increment
  )
  ```
  : we return a copy of the matrix `matrix`, where the value of each element is increased by the value of the `increment` parameter

-
  ```
  Matrix<element, height, width> operator*(
    const Matrix<element, height, width>& matrix,
    const element& factor
  )
  ```
  : we return a copy of the matrix `matrix`, where the value of each element is multiplied by the value of the `factor` parameter

-
  ```
  Matrix<element, height, width> operator+(
    const Matrix<element, height, width>& matrix1,
    const Matrix<element, height, width>& matrix2
  )
  ```
  : we add two matrices `matrix1` and `matrix2` of the same dimensions

-
  ```
  Matrix<element, height, width> operator*(
    const Matrix<element, height, depth>& matrix1,
    const Matrix<element, depth, width>& matrix2
  )
  ```
  : we multiply two matrices `matrix1` and `matrix2` of mutually compatible dimensions

Contrary to common practice, it is necessary to put the code of templated classes and functions in header files. Let us assume that in our case, there will only be one, and it will be named `Matrix.h`. Although it is not necessary in general, we will again consistently separate the definitions of both our classes (i.e., we will put the definition of the inner auxiliary class of the request outside of the body of the matrix class) as well as we will also separate definitions of all individual member functions of both our classes from their declarations (i.e., we will put their implementation outside of the body of the corresponding class). Because of that, it will become necessary to use the `typename` keyword in certain more complicated cases due to dependent names, in particular, when describing return types in our situation.

Submit only the aforementioned `Matrix.h` file. As usual, assume that the main file `Main.cpp` with the `main` function is already a part of the prepared test. The objective of this task is to get acquainted with the design of templated classes and functions, inner classes, the `std::array` array container, the `const_cast` retyping, as well as the implementation of further custom operators, namely arithmetic and the indexing operator.