NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C08**

Matrix

The goal of this task is to propose a class allowing for the representation of matrices containing elements of any numerical type, and subsequently implement several selected basic operations on such matrices.

In particular, we implement the matrix class as a class template `Matrix`, 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`), `Height` is the matrix height (number of rows), and `Width` its width (number of columns). Matrix elements, however, may not only be of primitive numeric types, we will also allow any suitable user-defined classes that implement the required behavior. Specifically, in such cases, we will expect an implicit constructor and additional operations such as stream insertion, increment, addition, or multiplication, if we choose to make use of them.

The internal storage for matrix elements is expected to be implemented via a private data member utilizing 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 individual elements. Specifically, let us assume index arithmetic where an element at logical coordinates `[row][column]` will be placed at a physical position `[row * Width + column]` within this internal array.

For a new matrix instance creation, we will 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 the specified sample element `value`. For basic work with our matrix, we offer the following member functions:

- `Element& get(size_t row, size_t column)`: returns a modifiable reference to an element placed at the indicated logical coordinates within a modifiable matrix, i.e., at row `row` and column `column`
- `const Element& get(size_t row, size_t column) const`: the same for a constant reference and constant matrix
- `void set(size_t row, size_t column, const Element& value)`: sets a 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 format: first, we put 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 matrix dimensions. Otherwise, the behavior of the respective functions will be undefined. We will therefore not perform any tests of input parameters, and full responsibility for the correct use of matrices is transferred to the caller. As for printing matrices, in order 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-increment and post-increment, 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 invoke the respective pre-increment or post-increment 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. Therefore, the semantics of these operations do not have to be the same and so we must not interchange them.

- `Matrix& operator++()`: performs matrix pre-increment, i.e., returns a reference to the existing matrix with the new element values already after their increment
- `Matrix operator++(int)`: performs matrix post-increment, i.e., returns 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 that 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, analogously to our `get` functions.

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 be provided with 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 `[]` operators on the first level (as methods on the matrix class itself) first return an instance of our auxiliary class, only then the `[]` operators on the second level (as methods on the request class) make 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
  ```
  ): returns 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
  ```
  ): returns 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>& matrix_1,
    const Matrix<Element, Height, Width>& matrix_2
  ```
  ): adds two matrices `matrix_1` and `matrix_2` of the same dimensions
- ```
  Matrix<Element, Height, Width> operator*(
    const Matrix<Element, Height, depth>& matrix_1,
    const Matrix<Element, depth, Width>& matrix_2
  ```
  ): multiplies two matrices `matrix_1` and `matrix_2` of mutually compatible dimensions

At the very end, we will adjust and extend the entire implementation of our matrix class in such a way that it supports the copy-on-write mechanism. This means we want to ensure that multiple instances of the matrix class can share the same content, and so the actual copying of matrices in terms of copying their data content only occurs at a moment when it becomes truly unavoidable. From the practical point of view, it is advisable to address this requirement only after you have already implemented and thoroughly debugged all the other matrix functionality. Fortunately, it will not be anything overly complicated.

The basic idea lies in the fact that the matrix class will no longer directly contain the `std::array` data member for the internal storage. Instead, it will only contain a smart pointer to it. Specifically, a shared smart pointer, which is designed precisely for such sharing functionality. The internal storage array will thus be allocated outside of the matrix class and will be created via dynamic allocation.

If we want to copy an existing matrix instance, in reality, only its wrapper with the smart pointer will be copied, while the internal storage will remain shared. This behavior is ensured automatically by the copy constructor and copy assignment operator for matrices as they are generated by the compiler. They just

blindly copy individual data members. In our case, this means copying the shared pointer, which leads to the sharing of its target. This is precisely what we need.

Whenever we invoke any operation on a matrix that directly leads to or potentially may lead to the modification of its content, we must ensure that, prior to the actual execution of such a modifying action, we first separate its data content. That is, if its internal storage is currently shared at that moment with one or even more matrices at a time, we create a new instance of the internal storage as a copy of the current content of the existing one. We will thus become its exclusive owner, and so we can proceed with the intended modifications, as the other originally interconnected matrices will no longer be affected by these modifications.

To ensure this mechanism, we propose an internal helper function in the matrix class. Its task will be to perform a test whether the separation is necessary and to perform it if required. For the test itself, we will use the `use_count()` method on a shared pointer. It returns the current number of active sharing participants. We will call the separation method within every matrix operation that performs any modifications. Let us not forget, however, that this also applies to methods that, while not performing any modifications by themselves, they return modifiable references to matrix elements. By providing such references, we are allowing the users to make modifications later without us being able to detect or treat them afterward at all. In other words, it would be too late.

We should also realize that by implementing the entire mechanism of the lazy behavior with deferred copying, we have caused the provided references to matrix elements to no longer have a permanent nature. This applies regardless of whether these are modifiable references or constant ones. In other words, any discussed modification action may invalidate them. We must be aware of these consequences and so perceive the obtained references merely as references intended for immediate use. Such behavior should not surprise us much, though, as in most standard containers, including, for instance, `std::vector`, invalidation of all references, pointers, or iterators may also occur because of performed modifications.

Contrary to the common practice, it is necessary to put the code of class and function templates 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 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 and use of class and function templates, 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.

Finally, let us have a look at the method for printing the content of a matrix, and the operations for a matrix pre/post-increment, adding a number to a matrix, multiplying a matrix by a number, adding two matrices, and, finally, multiplying two matrices. Without any doubt, we could straightforwardly implement them in a way that, using two nested loops (one for rows and the other for columns), we visit each individual position and process it accordingly, whether by calling our `get` or `set` functions or via the respective `[]` operators. However, this would not lead to particularly efficient code, not only due to the constant recalculation of the logical coordinates into the physical positions, but also due to the, let us say, chaotic traversal of the internal storage.

Although this is not required for the successful completion of this assignment, try to implement the above-mentioned functions in such a way that you eliminate both these drawbacks. That is, work directly with the internal private storage, traverse the elements stored in it sequentially, using a loop based on iterators. Use these iterators appropriately also when working with matrices of the respective operands.