C++ Programming

Exercise Sheet 7

Secure Software Engineering Group
Philipp Schubert
philipp.schubert@upb.de

June 04, 2021

Solutions to this sheet are due on 11.06.2021 at 16:00. Please hand-in a digital version of your answers via PANDA at https://panda.uni-paderborn.de/course/view.php?id=22691.
Note: If you copy text or code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited.

This exercise sheet will help to get a deeper understanding on how to use C++ in reality. For that reason, you will implement your own matrix template class type. You can achieve 16 points in total.

Exercise 1.
Implement a class template **matrix** that is able to store elements of an arbitrary type **T**. The core of your implementation will be **matrix operator* (const matrix& lhs, const matrix& rhs)**—a matrix multiplication. The matrix's entries are stored in a one dimensional vector to increase performance (you will learn why that is later on in this course). (Hint: For some function implementations you can, of course, make use of the functionalities of the **vector** type; you do not have to reinvent the wheel w.r.t. functionalities such as initialization.)

Consider the following interface:

```
#include <iostream>
#include <vector>
#include <initializer_list>
#include <stdexcept>

template<typename T>
class matrix {
 public:
  matrix(size_t rows, size_t columns);
  matrix(size_t rows, size_t columns, const T &ival);
  matrix(std::initializer_list<std::initializer_list<T>> imat);
  T& operator() (size_t row, size_t column);
  const T& operator() (size_t row, size_t column) const;
  size_t num_elements() const noexcept;
  size_t num_rows() const noexcept;
  size_t num_columns() const noexcept;
  friend matrix operator* (const matrix &lhs, const T &scale);
  friend matrix operator* (const matrix &lhs, const matrix &rhs);
  friend bool operator== (const matrix &lhs, const matrix &rhs);
```

```
  friend bool operator!= (const matrix &lhs, const matrix &rhs);
  friend std::ostream& operator<< (std::ostream &os, const matrix &m);

private:
  size_t rows;
  size_t columns;
  std::vector<T> data;
};
```

a) Start off by implementing the first two constructors, as well as the functions **num_elements()**, **num_rows()** and **num_cols()**. (1 P.)

b) Next, implement **operator<<** such that you can print variables of type **matrix** in a nice format to the command line. (1 P.)

c) Now, implement that odd-looking constructor, that constructs a matrix from a nested **std::initializer_list**. This constructor will be very helpful to construct variables of the matrix type using test data. (3 P.)

d) The above interface already contains one optimization: All matrix entries are stored in a one dimensional **stdvector**, that is, in one continuous block of dynamically allocated memory. We will learn why this is useful later on. But because the entries are stored in one dimension, one has to provide a function f that maps two dimensional matrix coordinates to the corresponding memory position in one dimension with $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. The mapping is defined as $f(x,y) \mapsto x \cdot c + y$ with $x \in \{0,1,..,rows-1\}$, $y \in \{0,1,..,cols-1\}$ and c the number of columns of the matrix. Overload **operator()** (the function call operator) to perform the mapping. Also implement the **const** version of that operator in order to be able to access elements from **matrix** variables that are declared **const**. (2 P.)

e) Justify why you do not have to implement the other special member functions! (1 P.)

f) Continue by implementing **operator==** and **operator!=**. Matrices should be considered equal if their dimensions and all of their entries are equal. (1 P.)

g) Implement **matrix operator* (const matrix& lhs, const double scale)** to return a matrix that is scaled by factor **scale** (this only works for numeric types, of course, see task j)). (1 P.)

h) Now, implement the core of this exercise **matrix operator* (const matrix& lhs, const matrix& rhs)** such that it performs a matrix multiplication—on numeric types (again, we leave it to the user of your matrix type to call both **operator*** on numeric matrices, see task j))—returning the resulting matrix. A matrix multiplication is defined as follows:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{12} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1p} \\ b_{12} & b_{22} & \dots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \dots & b_{mp} \end{pmatrix}$$

$$A \cdot B = \begin{pmatrix} (ab)_{11} & (ab)_{12} & \dots & (ab)_{1p} \\ (ab)_{12} & (ab)_{22} & \dots & (ab)_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (ab)_{n1} & (ab)_{n2} & \dots & (ab)_{np} \end{pmatrix}$$

with $(ab)_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj}$. In words: the resulting matrix contains scalar products of A's rows and B's columns. An illustration can be found in Figure 1. (3 P.)

i) Extend your implementation in task h) and check if both matrices have the correct dimensions for performing a matrix multiplication and if not, throw an adequate exception to inform the user of that type that the requested operation cannot be performed. (1 P.)

j) For both **operator*** (from task g) and task h)) use **type_traits** to check if the template parameter **T** is an arithmetic type such that the operations can be performed without crashing your program. Notify the user of your **matrix** type by using an adequate mechanism if **T** is not an arithmetic data type. (Hint: Have a look **std::is_arithmetic** defined in the **type_traits** header; also keep in mind that the early error is the better error.) (1 P.)

k) After having implemented the interface, do test your implementation by commenting-in the lines inside **main()**, compiling and running the ”test code”. Observe that you can measure runtimes of specific function calls by using the **chrono** header file as shown in the ”test code”. (1 P.)

l) This is an optional task: Let's pick up on task j). You are *the one*, you are one with the compiler and the machine: Instead of handling non-arithmetic types by issuing an error, for instance, if a user tries to multiply two matrices that store elements of type **std::string**, you completely disable both of the **operator*** for non-arithmetic type parameters by employing **std::enable_if**. Do so. (0 P.)

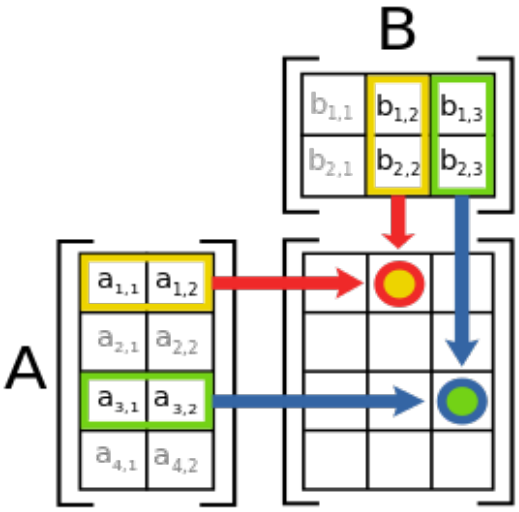

Figure 1: An illustration of a matrix multiplication.

Figure taken from wikipedia: https://en.wikipedia.org/wiki/Matrix_multiplication#/media/File:Matrix_multiplication_diagram_2.svg

Exercise 2.

Additional materials:

- Although C++ is the greatest language whatsoever, you always have to be critical with your software development tools. A really funny "WAT" talk summarizing some of the strangest properties of C++ (∼15 min) can be found here: `https://youtu.be/rNNnPrMHsAA`
- You may wish to deepen your current knowledge on C++ with help of the following talk: `https://youtu.be/86xWVb4XIyE` . Do not worry, we have not yet covered all of the topics of this talk.