C++ Programming

Exercise Sheet 07
Software Engineering, EIM-I
Philipp Schubert
December 1, 2017

Solutions to this sheet are due on 08.12.2016 til 14:00. Please hand in a digital version of your answers via e-mail. The e-mails subject has to contain cppp. Do zip-compress your solutions. For questions please send mail or speak to me during the exercises.

Note: If you copy text elements / code elements from other sources, clearly mark those elements and state the source. Copying solutions from other students is prohibited. All of your files that belong to your solution have to be contained in a single .zip file that is named according to the following naming scheme: <name>_<surname>_solution<XX>.zip. Replace <name> and <surname> with your actual name and replace <XX> with the sheet number the solutions belong to. You can look up your results using this link [https://docs.google.com/spreadsheets/d/1V8rKtimsQS6thKGkTh6CCh1v-LwulBIA3rvKAI7SH2M/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1V8rKtimsQS6thKGkTh6CCh1v-LwulBIA3rvKAI7SH2M/edit?usp=sharing)

During this exercise sheet you will learn more about how to use C++ in reality. For that reason, you will design/implement a simple matrix data type. You can achieve 16 points in total.

Exercise 1.

In this exercise you will implement a class template `matrix` that is able to store elements of an arbitrary type. The core of your implementation will be `matrix operator* (const matrix& lhs, const matrix& rhs)` that implements a matrix multiplication. The matrix entries are stored in a vector in one dimension to increase performance (we will learn why that is later on). (Hint: For some function implementations you can make use of the capabilities of type `vector`, you do not have to reinvent the wheel (e.g. initialization.).)

The code for this exercise can be found on the lectures website: [https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/WS2017_2018/code_07.zip](https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/WS2017_2018/code_07.zip)

Consider the following interface:

```cpp
#include <iostream>
#include <vector>
#include <initializer_list>
#include <stdexcept>
using namespace std;

template<typename T>
class matrix {
private:
  size_t rows;
  size_t columns;
  vector<T> data;
};
```
public:
  matrix(size_t rows, size_t columns);
  matrix(size_t rows, size_t columns, const T &ival);
  matrix(initializer_list<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 ostream& operator<< (ostream &os, const matrix &m);
};

a) Start off by implementing the first two constructors, as well as the functions number_elements(), number_rows() and number_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 initializer_list. This constructor will be very helpful in order to get test data in your matrix type. (3 P.)

d) The above interface already contains one optimization: All matrix entries are stored in one vector, that is, in one continuous block of 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 into the corresponding memory position in one dimension with f : N × N → N. The mapping is defined as f(x,y) = x·c + y where x ∈ {0,1,..,rows−1}, y ∈ {0,1,..,cols−1} and c the number of columns of the matrix. Implement operator() to perform the job. 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 as equal if both of their dimensions match and all their entries are the same. (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* only on numeric matrices, see task j) - returning the resulting matrix. A matrix multiplication works as shown below:

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

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

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

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 in order to perform a matrix multiplication, if not throw an adequate exception to inform the user of that type that the requested operation is not possible. (1 P.)

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

k) After having implemented the interface, do test your implementation by decommenting 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.)

Exercise 2.
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/rNNnPrMHaAA](https://youtu.be/rNNnPrMHaAA)

Exercise 3.
If you like, watch the following talk: [https://youtu.be/86xWVb4XIyE?t=7m18s](https://youtu.be/86xWVb4XIyE?t=7m18s) to deepen and extend your knowledge. Do not worry, we have not covered all of the topics in this talk just yet.
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