Dr. Ole Klein, Stefan Meggendorfer
IWR, Heidelberg University
Submission date 30.11.2021

Exercise Sheet 5

Exercise 1: Matrices and Templates (10 points)

In the lecture templates were presented as a technique of generic programming. They allow the development of algorithms independent of underlying data structures.

In this exercise we will extend an existing implementation of a matrix class to a template class. Such matrices are needed again and again in numerical software. On a previous sheet you had to implement matrices for `double` values. Depending on the application you want, however, it may be useful to have matrices based on other types of numbers, for example `complex`, `float` or `int`. Templates may reduce code redundancy, so that one has an implementation of `Matrix` that can be used for all the different types.

As a reminder, here is the functionality you had to implement:

- Matrix entries are stored as a vector of vectors, i.e., `std::vector<std::vector<double>>`.
- The matrix dimension is handled by two private `int` values, `numRows` and `numCols`.
- The parenthesis operator allows access to individual entries, while the bracket operator gives access to a whole row:

```
1 double& operator()(int i, int j);
2 std::vector<double>& operator[](int i);
```

- Free functions provide functionality for the multiplication of matrices with vectors and scalars, and for the addition of two matrices.

For your convenience, an implementation of such a matrix class can be found on the lecture website, based on the following files:

- `matrix.hh`
- `matrix.cc`
- `testmatrix.cc`

The first two contain the definition and implementation of `Matrix` for `double`, while the third file contains a test program. You may compile it like this:

```
1 g++ -std=c++11 -Og -g -o testmatrix matrix.cc testmatrix.cc
```

In addition to the methods specified in the previous exercise, this version also provides `const` versions of the parenthesis and bracket operators for read-only access, implements the free functions based on assignment operators `+=` and `*=`, provides `resize` methods, and in turn drops the second matrix-vector multiplication method.

You are free to use either your own version or the provided one as a basis for the following tasks. Do the following:

(a) Change the implementation of `Matrix` to that of a template class, so that it can be used for different number types.

(b) Turn `numRows` and `numCols` into template parameters. Remove any function that doesn't make sense under this change (e.g., `resize` if you use the provided version), adapt the constructors accordingly, and remove any runtime bounds checks that become unneccesary.

(c) Change the free functions into template functions that match the aforementioned template parameters. You may assume that scalars, vectors and matrices all use the same element type. Replace `std::vector` by a template template parameter, so that the function would also work for, e.g., `std::array`.

(d) Change the main function of the test program, so that your template variants are tested. The program should use all data types mentioned above, with `complex` referring to `std::complex<double>` and the required header being `<complex>`. Also test the matrix with your `Rational` class. For the `print` functionality you have to define a free function

```
std::ostream& operator<< (std::ostream& str, const Rational& r)
```

that prints the rational number by first printing its numerator, then a "/" and then its denominator.

Exercise 2: Matrices and Templates (II) (10 points)

This exercise is an extension of the templatization exercise above.

(a) Split the templated class `Matrix` into two classes:

- A class `SimpleMatrix`, without numerical operators. This represents a matrix containing an arbitrary data type and has the following methods, all taken from `Matrix`:
  - some constructors
  - `operator()(int i, int j)` for access to individual entries
  - `operator[](int i)` for access to whole rows
  - a `print()` method
- A numerical matrix class `NumMatrix` derived from `SimpleMatrix`. The numerical matrix class additionally provides the arithmetic operations functionality mentioned above.

Also modify the free functions accordingly.

(b) Write a test program that covers the two classes `SimpleMatrix` and `NumMatrix`. Your test program does not need to test each number type again, an arithmetic test with `double` is enough. Instead, test your `SimpleMatrix` with `string`s as data, and check whether you can create and use objects of type `SimpleMatrix<NumMatrix<...>,...>` and `NumMatrix<NumMatrix<...>,...>`. Which methods are you able to use? What happens when you try to call the `print` method?

The latter matrix type is known as a block matrix, a type of matrix that arises in numerical discretizations and can be used to design efficient algorithms if it has additional structure.

(c) Write template specializations of `SimpleMatrix` so that print functionality is restored for block matrices. Block matrices should be printed as one big matrix, with "|" as horizontal separators between matrix blocks and rows of "---" to separate the blocks vertically (it is okay if it isn't the most beautiful output). *Note:* it might be helpful to assign the dimensional template parameters to public enums, so that you have the dimensions available during printing. What do you observe when implementing this?

This would be a rather good opportunity to use SFINAE to provide two different `print` methods and choose the right one based on the template parameters, instead of having to provide several different base classes.

(d) The provided matrix class (and most likely also your own) simply aborts the program when matrix-vector products with mismatching dimensions are attempted. Replace the relevant checks with adequate exception handling. Derive your own custom exception from `std::exception`, or pick a fitting one from the predefined ones if applicable. Add a try/catch block to your test that triggers an exception and prints an explanatory message.