Dr. Ole Klein, Stefan Meggendorfer
IWR, Heidelberg University
Submission date 16.11.2021

Exercise Sheet 3

Exercise 1: Matrix Class Implementation (20 points)

Note: this exercise will form the foundation for several of the upcoming exercises, so if you want to work on only a subset of the sheet, it might be a good idea to start on this one.

In Scientific Computing the concept and use of matrices is crucial. Regardless of the field of expertise — if it is in optimization, statistics, artificial intelligence, or the solution of partial differential equations — we need matrices and solutions of linear systems of equations in nearly all applications.

In this exercise, we will implement a class `Matrix` in analogy to the class `Vector` from exercise sheet 1.

Take care of the following points:

(a) Class `Matrix` should have all functionality that class `Vector` has (constructors, methods, etc.).

(b) The entries should be stored in a container of type `std::vector<std::vector<double>>`.

(c) Instead of the number of elements `int N` in class `Vector`, class `Matrix` should contain the two numbers `int numRows` and `int numCols` that represent the number of rows and the number of columns respectively. Use `numRows` and `numCols` in all places where the member functions of class `Vector` used `N`, and re-implement the functionality adapted to the use of a class that represents matrix objects.

(d) Use the member function `double& operator()(int i, int j)` for accessing the (i, j)-th element of objects of class `Matrix`.

(e) Use the member function `std::vector<double>& operator[](int i)` to return the i-th row of a matrix object.

(f) Class `Matrix` should have an additional constructor that constructs square matrices.

In addition to the member functions mentioned above, implement free functions that provide

(g) the addition of two matrices,

(h) the multiplication of a matrix with a scalar,

(i) the multiplication of a scalar with a matrix,

(j) a matrix-vector multiplication, where vectors are of type `std::vector<double>`,

(k) a matrix-vector multiplication, where vectors are of type `Vector`.

Write a test program that tests all functionality of class `Matrix` (construction, the different kinds of multiplication, element access).

Exercise 2: Linked List (10 points)

Using the simple example of a chained list we will practice the interaction of constructors, destructors and pointers.

We want to program a linked list which can store an arbitrary number of values of type `int`. Such a list consists of an object of class `List`, which refers to a sequence of objects of class `Node`. The list elements are stored in a component `int value` within each node, and a pointer `Node* next` points to the next node. The end of the list is designated by the pointer `next` having the value `nullptr`.

(a) What is special about a pointer having the value `nullptr`?

(b) Implement the class `Node`. Make sure that all member variables are always initialized, especially the `next` pointer.

(c) Implement the class `List` with the following methods:

```
class List
{
  public:

    List ();                          // create an empty list
    ~List ();                         // clean up list and all nodes
    Node* first () const;             // return pointer to first entry
    Node* next (const Node* n) const; // return pointer to node after n
    void append (int i);              // append a value to the end
    void insert (Node* n, int i);     // insert a value before n
    void erase  (Node* n);            // remove n from the list
};
```

`List` must also store the beginning of the list, where would you place it in the class declaration? The `next` pointer of class `Node` should be `private` to ensure that the list structure isn't accidentally changed outside of class `List`. The member `value` is `public` to allow read and write access from outside the class. The line `friend class List;` has to be inserted into the declaration of the class `Node` to give the `List` class access to the `next` pointer. Additionally make sure that the destructor deletes all allocated `Node` objects.

(d) Test your implementation with the following program:

```
int main ()
{
  List list;
  list.append(2);
  list.append(3);
  list.insert(list.first(), 1);

  for (Node* n = list.first(); n != 0; n = list.next(n))
    std::cout << n->value << std::endl;
}
```

(e) What happens if one copies the list? And what happens if both lists are deleted?

```
int main ()
{
  List list;
  list.append(2);
  ...
  List list2 = list;
}
```