Dirk Praetorius,
Gregor Gantner

Sommersemester 2018
19.06.2018

Übungen zur Vorlesung Einführung in das Programmieren für TM

Serie 12

Aufgabe 12.1. Implement the class `Person` which contains the members `name` and `address`. Derive from `Person` the class `Student`, that contains the additional data fields `matriculationNumber` and `study`. Derive from `Person` also the class `Worker` that contains the additional data fields `salary` and `work`. Write `set/get` functions, constructors, and destructors for all classes. Moreover, write a main progam to test your implementation!

Aufgabe 12.2. Implement the method `print` for the basis class `Person` from Exercise 12.1. The method should print to the screen name and address of a person. Redefine this function for the derived classes `Student` and `Worker` so that also the additional data fields of these classes are printed. Moreover, write a main programm to test the `print`-methods of the different classes.

Aufgabe 12.3. Derive the class `SquareMatrix` from the class `Matrix` from the lecture. This class is used to store square matrices and should contain all functionalities from the basis class `Matrix`. Test your implementation accurately!

Aufgabe 12.4. We consider the class `Matrix` as well as the derived classes `Vector` from the lecture and `SquareMatrix` from Exercise 12.3. Implement the method `solve` for the class `SquareMatrix`, which solves the linear system $Ax = b$ by using the so-called *Gaussian elimination.* Consider a matrix $A \in \mathbb{R}^{n\times n}$ (type `SquareMatrix`) and a right-hand side vector $b \in \mathbb{R}^n$ (typ `Vector`). The algorithm reads as follows:

- First of all, the matrix A is converted into an equivalent upper triangular matrix. Note that also the right-hand side vector b must be modified accordingly.
- The resulting system, characterized by an upper triangular matrix A, is then solved directly.

In particular, during the first elimination step, an appropriate multiple of the first row of the matrix is subtracted from the remaining rows so to obtain a matrix of the form

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2} & \dots & a_{nn} \end{pmatrix}.$$

In the second elimination step, an appropriate multiple of the second row of the matrix is subtracted from the remaining rows so to obtain a matrix of the form

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{n3} & \dots & a_{nn} \end{pmatrix}.$$

After $n-1$ elimination steps, one obtains an upper triangular matrix A. Use `assert` to ensure that, in the k-th elimination step, the condition $a_{kk} \neq 0$ is satisfied. Don't forget that also the right-hand side vector $b \in \mathbb{R}^n$ must be modified accordingly. To solve the system $Ax = b$ with A being an upper triangular matrix, use the same approach considered for lower triangular matrices in Exercise 11.5). What is the computational cost of your implementation of the Gaussian elimination and why? To undestand the algorithm, start with simple examples with $A \in \mathbb{R}^{2\times 2}$ and $A \in \mathbb{R}^{3\times 3}$. Test your implementation accurately!

Aufgabe 12.5. The Gaussian elimination algorithm from Exercise 12.4 fails when it happens that $a_{kk} = 0$ in the k-th elimination step. This can happen even when the linear system $Ax = b$ has a unique solution x. To avoid this, the algorithm is usually extended with the so-called *pivoting*:

- During the k-th step, choose amongst $a_{kk}, \ldots, a_{nk}$ the element a_{pk} with the largest absolute value.
- Swap the k-th and the p-th row of A (and b).
- Perform the elimination step as before.

Implement for the class `SquareMatrix` from Exercise 12.3 the method `gausspivot`, which computes the solution of the system $Ax = b$ following the aforementioned strategy. (It is possible to prove that the Gaussian elimination algorithm with pivoting can be successfully applied if and only if the linear system $Ax = b$ admits a unique solution. A proof of this result can be found in any numerical analysis book. Test your implementation accurately!

Aufgabe 12.6. Derive from the class `SquareMatrix` from Exercise 12.3 the class `DiagonalMatrix`. Only the diagonal entries of the matrix must be stored. Implement constructors, type cast and access to the coefficients. For each entry A_{ij} with $i \neq j$ proceed as follows: Save additional private members `double zero` and `double const_zero` and use it to access the coefficients. That means that a call of the `()`-operator for `const`-objects returns `const_zero` and in case of a normal call the `()`-operator returns `zero`. Make sure to set `zero` to 0 in every non-`const` call of the `()`-operator. Why? Test your implementation accurately!

Aufgabe 12.7. Redefine the method `solve` from Exercise 12.4 for the class `DiagonalMatrix` from Exercise 12.6 in such a way that the system $Ax = b$ with A being a diagonal matrix is solved by exploiting the diagonal structure of the matrix. What is the computational cost of your implementation and why? Test your implementation accurately!

Aufgabe 12.8. What is the output of the following programme? Explain why!

```
#include <iostream>
using std::cout;
using std::endl;
class BasisClass {
   protected:
      int N;
   public:
      BasisClass() {
         N = 0;
         cout << "Standard constr. BasisClass" << endl;
      }
      BasisClass( int n) {
         N = n;
         cout << "Constr. BasisClass, N = " << N << endl;
      }
      ~BasisClass(){
         cout << "Destr. BasisClass, N = " << N << endl;
      }
      BasisClass( const BasisClass& rhs) {
         N = rhs.N;
         cout << "Copy constr. BasisClass" << endl;
      }
      BasisClass& operator=(const BasisClass& rhs) {
         N = rhs.N;
         cout << "Assignment operator BasisClass" << endl;
         return *this;
      }
      int getN() const { return N; }
```

```
        void setN( int N ) { this->N = N; }
};

class Derived : public BasisClass {
    public:
        Derived(){
            cout << "Standard constr. Derived" << endl;
        }
        Derived( int n):BasisClass(n) {
            cout << "Constr. Derived, N = " << N << endl;
        }
        ~Derived() {
            cout << "Destr. Derived, N = " << N << endl;
        }
        Derived( const Derived& rhs) {
            N = rhs.N+7;
            cout << "Copy constr. Derived" << endl;
        }
        Derived& operator=(const Derived& rhs) {
            N = rhs.N;
            cout << "Assignment operator Derived" << endl;
            return *this;
        }
};

Derived foo(Derived X){
    Derived tmp(5);
    tmp.setN(X.getN()*X.getN());
    return tmp;
}

int main() {
    Derived ah(10);
    {
        Derived gg(13);
        BasisClass bs;
        BasisClass mr=bs;
        ah=gg;
    }
    ah=foo(ah);

    return 0;
}
```