Solutions to this sheet are due on 17.11.2016 til 14:00. Please hand in a digital version of your answers via e-mail. The e-mails subject has to contain ccpp. 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>_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/1V8rKtimsQS6thKGkTh6CClv-LwsBI3RvKA1ZSH2M/edit?usp=sharing

During this exercise sheet you will learn how to use dynamically allocated memory and operator overloading. Additionally, you will implement a small useful algorithm. You can use the code snippets from https://www.hni.uni-paderborn.de/fileadmin/Fachgruppen/Softwaretechnik/Lehre/CPP_Programming/WS2017_2018/code_04.zip. You can achieve 16 points in total.

Exercise 1.

In this exercise you will create a simple model of a mathematical vector $v \in \mathbb{R}^n$ in order to make yourself familiar with dynamic memory allocation and operator overloading. This time you will not use std::vector to store the elements but rather create your own data type. The STL data type std::vector that you have already used, is implemented in a very similar manner to what you are going to implement in this exercise. Consider the code provided on the website, all (special member) functions signatures are annotated with comments that describe what each function should do. Provide implementations for all (special member) functions and test your implementations by uncommenting the test code provided in the main function. (Hint: Have a look on how we implemented the special member functions in the lecture.)

a) Provide implementations for the following useful constructors:

- vec(size_t size);
- vec(size_t size, double ival);
- vec(initializer_list<double> ilist); (look up std::initializer_list on en.cppreference.com)

(3 P.)
b) Furthermore, provide implementations for the following other special member functions.
   - `~vec();`
   - `vec(const vec &m);`
   - `vec(vec &&m);`
   - `vec& operator=(const vec &m);`
   - `vec& operator=(vec &&m);`

(3 P.)

c) Also provide implementations for the following useful functions.
   - `size_t size();`
   - `double& at(size_t idx);`
   - `const double& at(size_t idx) const;`
   - `double& operator[](size_t idx);`
   - `const double& operator[](size_t idx) const;`
   - `friend ostream& operator<<(ostream &os, const vec &v);`
   - `friend vec operator+ (vec lhs, const vec &rhs);`
   - `friend vec operator- (vec lhs, const vec &rhs);`
   - `friend vec operator* (vec lhs, double scale);`
   - `friend double operator* (const vec &lhs, const vec &rhs);`

(5 P.)

Exercise 2.

In this exercise you will learn about sorting. Bubble sort is a sorting algorithm that allows e.g. the sorting of a `std::vector`. The way bubble sort works is that it iterates a `vector v` and looks at two adjacent elements `v[i]` and `v[i + 1]`. Then bubble sort compares these two elements and swaps their position if the value `v[i + 1]` is smaller than `v[i]`. It then increments `i` and performs the next "bubble" comparison until it has iterate through the complete `vector`. One iteration might not be sufficient to sort all entries of `v`. Therefore, bubble sort performs as many iterations as needed until nothing has to be swapped anymore; the `vector` is then sorted.

a) Implement a function `void bubble_sort(vector<int> & v)` that sorts the vector of integers specified by the reference according to the bubble sort algorithm. Your implementation has to sort all entries in `v` in ascending order (small numbers first, described as above). Test your bubble sort implementation for the following `vector` variable: `vector<int> v = {1, 5, 6, 23, 7, 8, 9, 21, 12, 4};`

(3 P.)

b) Modify your bubble sort implementation to match the following signature `void bubble_sort(vector<int> & v, size_t from, size_t to)` and change its behavior such that it only sorts the entries that are contained in the interval specified by `from` and `to`. For example the following call `bubble_sort(v, 0, 5)` would result in changing `vs` content to `1, 5, 6, 7, 8, 23, 9, 21, 12, 4`.

(2 P.)