Exercise 1.
In this exercise, you will create a simple model of a mathematical vector \( \mathbf{v} \in \mathbb{R}^n \) 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. Consider the code provided on the website, all (special member) function 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(initilizer_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.

- \( \sim \text{vec}() \);
- \( \text{vec(const vec \&m)} \);
- \( \text{vec(vec \&\&m)} \);
- \( \text{vec\& operator= (const vec \&m)} \);
- \( \text{vec\& operator= (vec \&\&m)} \);

(3 P.)

c) Also provide implementations for the following useful functions.

- \( \text{size\_{t} size();} \);
- \( \text{double\& operator[] (size\_{t} idx);} \);
- \( \text{const double\& operator[] (size\_{t} idx) const;} \);
- \( \text{friend ostream\& operator<< (ostream \&os, const vec \&v);} \);
- \( \text{friend vec operator+ (vec lhs, const vec \&rhs);} \);
- \( \text{friend vec operator- (vec lhs, const vec \&rhs);} \);
- \( \text{friend vec operator* (vec lhs, double scale);} \);
- \( \text{friend double operator* (const vec \&lhs, const vec \&rhs);} \);

(5 P.)

Exercise 2.
This exercise is about sorting. Bubble sort is a sorting algorithm that allows you to sort the elements of a `std::vector`, for instance. 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 iterated 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 a vector of integers specified by the reference parameter according to the bubble sort algorithm. Your implementation has to sort all entries in `v` in ascending order (small numbers first, as described in the above). Test your bubble sort implementation for the following `vector` variable:

\[
\text{vector<int> v = \{1, 5, 6, 23, 7, 8, 9, 21, 12, 4\};}
\]

(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 change `v`'s content to `1, 5, 6, 7, 8, 23, 9, 21, 12, 4`.

(2 P.)