Exercise 1.
In this exercise, you will create a simple model of a mathematical vector $v \in \mathbb{R}^n$ 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 vector-like data type that uses dynamic memory allocation to store its elements. The STL data type std::vector that you already used is implemented in a very similar manner to what you will implement in this exercise. **Consider the code provided on the website, all (special member) function signatures are annotated with comments that describe what each function must do.** Provide implementations for all 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 at [en.cppreference.com](https://en.cppreference.com))
   (3 P.)
b) Furthermore, provide implementations for the following other special member functions.

- `~vec();`
- `vec(const vec &m);`
- `vec& operator=(const vec &m);`
- `vec(vec &m);`
- `vec& operator=(vec &m);`

(4 P.)

c) Also provide implementations for the following useful function members and operators.

- `size_t size();`
- `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);`

(4 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. Here is how bubble sort works: it iterates a `std::vector`-typed variable `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 `std::vector`. One iteration might not be sufficient to sort all entries of `v`. Therefore, bubble sort performs as many iterations as necessary until nothing has to be swapped anymore; the `std::vector` variable 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 `std::vector` variable:

```c++
std::vector<int> v = {1, 5, 6, 23, 7, 8, 9, 21, 12, 4};
```

(3 P.)

b) Modify your bubble sort implementation to match the 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 contents to `1, 5, 6, 7, 8, 23, 9, 21, 12, 4`. (2 P.)