Exercise 6.1 – Template functions

- The goal of this exercise is to rewrite the sorting function of Exercise 2.1 as a generic algorithm by writing it as a *template* function.
  a) Start from the solution for Exercise 2.1 that is based on passing references to integers as a starting point for this exercise.
  b) First, turn the `order()` function in to a *template* function by prefixing the function declaration with a `template<class T>` line. Note that both the declaration and the definition of the function need to be prefixed in this way. Now change all occurrences of the type `int` in `order()` with the new generic type `T`. Hint: There are 3 occurrences in total
  c) Next, turn the `sort()` function into a *template* function in the same way: prefix the function declaration and definition with a template declaration and replace all occurrences of type `int` with the generic type `T`.
  d) Compile the program and verify that it works.
  e) Move the code fragment in `main()` that prints the contents of the array into a `display(T t[], int n)` function and make `display()` a template function as well.
  f) Verify once more that everything works for arrays of integers. Now add an array of randomly ordered float values to your `main()` function and `sort()` and `display()` them as well. If your code is properly written a version of `sort()` and `display()` that works with floats should be generated automatically by the compiler.

Exercise 6.1 – Template functions

g) Finally, add an array `char*` pointers to `main()` (You can reuse the list you created in exercise 2.1 for the sorting of strings) and try to `sort()` and `display()` them as well.

h) You will see that the above code does not work, try to understand why.

i) Fix the string sorting problem by introducing a template specialization for type `char*` that replaces the regular comparison operator`(<)` with a `strcmp()` function call, similar to your string sorting solution of exercise 2.1. Think about which function do you need to specialize: `order()`, `sort()` or both?

Exercise 6.2 – A generic container class

- The goal of this exercise is to write a class Array that mimics the behavior of a C++ array, but provides more intelligent memory management
  a) Start with the input class `ex6.2/Array.hh` that implements a simple implementation of an array of double values. Convince yourself that the class correctly implements the constructor, destructor, copy constructor and assignment operator.
  b) Change the `class Array` into a `template class Array`.
  c) Use class `Array` in a small test program to store an array of integers and to store an array of '`const char*`' strings.
  d) Does the code in `operator[]` look safe to you? What happens if you try to access element 1000 of an array of length 10?
  e) Change the `operator[]` such that when an element beyond the range of array is accessed, the array is automatically extended to include that element using the `resize()` function.
  f) A drawback of the `resize()` operation is that the newly created elements do not have a defined value. Change the constructor of class `Array` such that it has takes two arguments: the initial capacity and the default value that is assigned to 'blank' elements. Change the `resize()` function such that it uses the specified default value to initialize all newly allocated elements, and make sure that the default values are transferred in the copy constructor and the assignment operator.

Exercise 6.3 – Revisiting the class Stack

- The goal of this exercise is to revisit the Stack class from exercise 3.3 and revisit its storage strategy from a 'raw' C++ array into the use of the 'smart' class Array.

  a) Copy the provided solution for the `Stack` class from exercise 3.3, its main program as well as the `Array` class from exercise 6.2. Compile the code and verify that works OK.

  b) We will now rewrite class `Stack` to use class `Array` for internal storage. We start with the data members of class `Stack`: In the solution of Ex. 3.3, the data is stored using an array of doubles (`double *s`) and an associated length (`int len`). Replace these with an `Array<double> s`. Don't forget to include the `Array.hh` header file in `Stack.hh`.

  c) There are several places in the code of `Stack` that use the length of the internal memory buffer that used to be stored in `len`. This information is now available from `Array<double> s`, so replace each occurrence of `len` in the code with `s.size()`, which reports the size of the buffer in `s`.

  d) Do you still need an explicit copy constructor and destructor for class `Stack`? Hint: do you still have pointers to 'owned' memory as data members?. If not, remove them.

  e) Adapt the constructor to initialize all data members: `s` with given size and `counter` with value zero. Eliminate function `init()` since it is now superfluous.

Exercise 6.3 – Revisiting the class Stack

f) Remove `grow()` entirely since its functionality is now mostly absorbed in `Array`. In `push()` replace `grow()` with `s.resize(s.size()+10)`.

g) Test the modified `Stack` class with the main program to verify that it still works.

h) As a final step, turn `class Stack` into a `template class Stack`. To do so you need to change any reference to type `double` to a generic type `T` in the code and add a `template<class T>` line to the class declaration.