Exercises:

1) General questions:
   a) What <u>exactly</u> does the operator sizeof return?
2) Get acquainted with the usage of dynamic memory. Learn how your IDE reacts <u>within the debugger</u> and <u>without the debugger</u> if:
   a) you create a block of dynamic memory and <u>exit the program w/o freeing</u> that memory block.
   b) you create a block of dynamic memory and <u>free it twice</u>.
   c) you <u>free a pointer to an automatic variable on the stack</u>.
   Document the results (use screenshots if required).
3) Show that subsequent local automatic variables have decreasing addresses on the stack. Use a program with console output to show this.
4) Implement a function that accepts a cstring and returns a new cstring, which contains the content of *a* but with all upper case letters.
5) Write a new program. It should ask the user for words, as long as he enters quit, the program should then send the words to stdout, but no word should be send twice!
6) Implement a function that accepts an int parameter (named *length* for example) and returns an int-array. The returned array should be created on the heap with the size length is passed as argument (in the parameter *length*). A program should present the functionality.
7) Implement a function called *trim()*, which trims the passed cstring's whitespaces and returns a new cstring.
8) Modify the "sum of squares" once again. This time a new option in the menu should be implemented, [count of numbers], which allows the user to specify the count of numbers to be processed. If this option was not used by the user, the count should default to five. This time you need to use a dynamically allocated array. Be careful to allocate enough memory for the program's operations and to free the allocated memory correctly.
9) Create a cstring array in the heap, which has space to hold four cstrings. Then create the four cstrings "Nina", "Lisa", "Greta" and "Mary" in the heap and assign them to the formerly created array's elements. Print the four cstrings stored in the array on the console. Finally free all the allocated heap memory.
10) Let's revisit the array-copy function from the last bunch of exercises ((4) Exercises, 3).
    a) Learn what the function *std::memcpy*() does and how it should be used.
    b) Rewrite the function, so that it uses the function *std::memcpy()* to copy the content from one array to another array.
    c) Write your own version of *std::strcpy()* in terms of *std::memcpy()*.
    d) Rewrite the test program so, that it provides a menu to the user. The user can specify the size and the content of the source array and after the array-copy function returns the content of the destination array should be printed to the console.
    e) Which error cases do you have to take into consideration in your test program?
11) Create a program to which the user has to pass some words via the <u>command line</u>. Internally, the function *main()* should call another function that accepts the command line arguments in order to process them. The first argument (i.e. the name of the program) should <u>not</u> be passed to that function. The discussed function should return an array of cstrings containing the input command line arguments in upper cases (i.e. the function converts the input cstring array into an array of upper case cstring). The function *main()* should then print the result.

Example:

```
> Program.exe test gaga zulu
> TEST GAGA ZULU
```

12) Write a program, which creates a memory leak deliberately. Show that there really is a memory leak with the means of the operating system (OS).
13) Program the hangman game.

Remarks:

- If exercises ask to document something, a Word document with explanatory text, maybe incl. snippets and screenshots is awaited as companion artifact in the repository or sent as attachment to the solution of the exercise!
- When writing functions, apply separated compilation, i.e. separate h-files from cpp-files!
- The functions must have documentation comments, e.g. following the HeaderDoc convention.
- Everything that was left unspecified can be solved as you prefer.
- In order to solve the exercises, only use known constructs, esp. the stuff you have learned in the lectures!
- **Please obey these rules for the time being:**
  - **When using g++ (e.g. via Xcode), your code must successfully compile with the -pedantic compiler-flag, which deactivates any non-standard C++-extensions.**
  - **The usage of goto, C++11 extensions, as well as #pragmas is not allowed.**
  - **The usage of global variables is not allowed.**
  - **You mustn't use the STL, esp. *std::string*, because we did not yet understood how it works!**
  - **But *std::cout*, *std::cin* and belonging to manipulators can be used.**
  - **You mustn't use new and delete!**
  - **You are not allowed to use C++ references instead of pointers.**
- Avoid magic numbers and use constants where possible.
- The results of the programming exercises need to be runnable applications! All programs have to be implemented as console programs.
- The programs mustn't have any memory leaks!
- You should be able to describe your programs after implementation. Comments are mandatory.
- In documentations as well as in comments, strings or user interfaces make correct use of language (spelling and grammar)!
- Don't send binary files (e.g. the contents of debug/release folders) with your solutions! Do only send source and project files.
- Don't panic: In programming multiple solutions are possible.
- If you have problems use the Visual Studio help (F1) or the Xcode help, books and the internet primarily.
- Of course you can also ask colleagues; but it is of course always better, if you find a solution yourself.