Exercises:

1. Define an instance of type int in the heap (*std::malloc()*) and create an instance of type int in the freestore (new). Initialize the int created on the freestore with the value 42.
2. Write a program that proves that an instance of UDT gets copied when it is passed to a function by value.
3. A new class should be implemented. It should have the name *DynamicIntArray* and it should simulate the functionality of a dynamically growing array and it encapsulates an ordinary C/C++ int-array.
   a) *DynamicIntArray* has following interface:
      1. After a new instance of *DynamicIntArray* was created with the dctor, the encapsulated array should contain ten elements. All elements should be initialized with the value 0.
      2. An additional ctor accepting an int parameter should initialize the encapsulated array with the argument's size (e.g. different from the default size of ten elements). All elements should be initialized with the value 0.
      3. The member functions *GetElementAt()*/*SetElementAt()* access and modify the existing elements.
      4. Only the member functions *AddElement()* can add new elements by appending them to the array. – If the encapsulated array is too small it needs to be enlarged.
      5. The member function *GetSize()* returns the count of items in the array.
   b) Comment the API, so that users can understand and use *DynamicIntArray* and its API, w/o inspecting the code.
   c) Implement RAII.
   d) Create tests for this API! Mind error cases!
   e) Create an application to let the user play with *DynamicIntArray* (incl. a menu).
4. Create a class *File* which encapsulates an *std::FILE* (<cstdio>) and allows writing a file. You'll have to learn about *std::FILE*'s API to complete this exercise. The class should have at least following member functions:
   a) A ctor accepting a cstring that should represent the name of the file.
   b) A member function *File::Write()* accepting a cstring containing the text to write to the file.
   c) A member function *File::Close()* closing the encapsulated *std::FILE*.
   d) The dtor should do the clean up.
   e) Comment the API, so that users can understand and use *File* and its API, w/o inspecting the code.
   f) Create tests for this API! Mind error cases!
   g) Create an application to let the user play with *File* (incl. a menu).
5. Create following static member functions for the type *File*:
   a) *File::Exists()* accepting a cstring containing file name to check for existence.
   b) *File::Rename()* accepting a cstring containing file name to rename.
   c) *File::Remove()* accepting a cstring containing file name to remove.
   d) Create tests for the updated API! Mind error cases!
   e) Update the created application to let the user play with the updated API.
6. Learn about the type *std::auto_ptr*.
   a) What has *std::auto_ptr* to do with RAII?
   b) Program an example, in which std::auto_ptr is used. – What benefits do we get?
   c) Why is *std::auto_ptr* deprecated? (Show an example!) – C++11 introduced a new type to replace *std::auto_ptr*, how does this type work? (Show an example!)

Remarks:

- 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:**
  - **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.**
- Only use classes for your UDTs. The usage of public fields is not allowed! The definition of inline member functions is not allowed!
- Do not put class definitions and member function definitions into separate files (we have not yet discussed separated compilation of UDTs).
- The results of the programming exercises need to be runnable applications! All programs have to be implemented as console programs.
- The programs need to be robust, i.e. they should cope with erroneous input from the user.
- Stick to the agreed upon coding conventions.
- 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.