Car register I

Submission deadline:	**2021-03-28 23:59:59**	Zbývá **1 dní, 10h, 45m a 42s**
Late submission with malus:	**2021-06-30 23:59:59** (Late submission malus: 100.0000 %)	
Evaluation:	**0.0000**	
Max. assessment:	**5.0000** (Without bonus points)	
Submissions:	0 / 20 Free retries + 20 Penalized retries (-2 % penalty each retry)	
Advices:	0 / 2 Advices for free + 2 Advices with a penalty (-10 % penalty each advice)	

Your task is to implement class `CRegister`, which implements a car database.

The database is very simplified. We assume the database saves information about persons and the cars they own. Each person is described by his/her name and surname. We assume the pair (name, surname) is unique in the database. That is, there may exist many persons with the same name (e.g. "John"), there may be many persons registered with the same surname (e.g. "Smith"), however, there may be at most one "John Smith". Each car is identified by its license plate, the plate is unique. Finally, each car must be owned by exactly one person, however, a person may own an arbitrary number of cars.

The class shall implement public interface shown below. The interface consists of:

- A constructor (without parameters). This constructor initializes a new empty database.
- Destructor -- it shall free all resources used by the instance.
- Method `AddCar (rz, name, surname )` which adds a new record to the database. The parameters are `name` and `surname` of the person being registered. Parameter `rz` is the license plate of the registered car. The method returns `true` if it succeeds, or `false` if it fails (i.e. the `rz` is already present in the database).
- Method `DelCar (rz)` removes the corresponding record from the database, the record is identified by the registration plate. The method returns `true` if it succeeds, or `false` if it fails (the corresponding record was not present). In addition to the car, the method may need to delete the person if it deleted the last car the person owned.
- Method `Transfer (rz, name, surname)` modifies the database such that the original owner of car `rz` sold the car and the car now belongs to person identified by the name and surname. The method returns `true` if it succeeds, or `false` if it fails (the corresponding car was not found or the seller and the buyer is the same person). Finally, the method erases the previous owner of the car if that owner does not have any other car.
- Method `CountCars (name, surname)` count the cars owned by person identified by the name and surname. If the person is unknown, the result is zero.
- Method `ListCars ( name, surname)` lists the cars owned by the person identified by the name and surname. The result is an object of class `CCarList`. The object represents a list of cars that belong to the person. The object may be used to iterate through the list to access the individual cars (see below). If the person does not exist/is unknown, the resulting object represents an empty list.
- Method `ListPersons` gives access to the persons in the register. It returns an object of class `CPersonList`, the object may be used to list the names and surnames of all persons in the database. If the database is empty, the resulting object represents an empty list.

Class `CCarList` represents a list of cars. The object may be used to iterate through the records and to obtain the registration plates of individual cars. The interface is:

- `RZ` - return the registration plate of the active car in the list,
- `AtEnd` - indicate whether we reached the end of the list (returns true), or not, i.e. there are further cars available (returns false),
- `Next` - move one record forward in the car list.

Class `CPersonList` is used to iterate over the existing persons in the database. Exactly one person is accessed at a time, the object offers an interface to move forward in the list op persons. The persons are accessed in a sorted order - sorted primarily by surname, then (for equal surnames) sorted by names. The interface is:

- method `Name` - retrieve the name of the currently accessed person,
- method `Surname` - retrieve the surname of the currently accessed person,
- method `AtEnd` - indicate whether we reached the end of the list (returns true), or not, i.e. there are further persons available (returns false),
- method `Next` - move one record forward in the person list.

Submit a source file with your `CRegister`, `CCarList` and `CPersonList` implementation. The classes must follow the public interface above. If there is a mismatch in the interface, the compilation will fail. You may extend the interface and add you auxiliary methods and member variables (both public and private, although private are preferred). The submitted file must include both declarations as well as implementation of the classes (the methods may be implemented inline but do not have to). The submitted file shall not contain `main`, your tests, or any #include definitions. If present, please, keep them in a conditional compile block. Use the attached template a basis for your implementation. If the preprocessor definitions are preserved, the file may be submitted to Progtest.

The class is tested in a limited environment -- both memory and running time is limited. The available memory is big enough to store the records. The `CRegister` class does not have to implement a copy constructor or an overloaded operator =. This functionality is not tested in this homework. Moreover, the copy functionality is explicitly disabled in the attached template (declarations `... = delete`). It is a good idea to keep these declarations in place.

On the other hand, both iterators `CCarList` and `CPersonList` are copied. Indeed, they are copied as soon as they are created in `CRegister::ListCars` and `CRegister::ListPersons` methods. Their implementation may be simplified. You may rely on the the contents of the originating `CRegister`, the instance is not destroyed nor modified until the iterator reaches the end. Therefore, the iterators may reference the data in the original `CRegister` instance without the need to copy them.

Your implementation must be both time and space efficient. A simple linear-time solution will not succeed (it takes more than 5 minutes for the test data). You may assume car transfers, car listing and counting owned cars is significantly more frequent than adding / removing cars.

Either STL container or dynamic array allocation is required to implement the database. If implemented using the dynamically allocated array, set the initial size of the array to some small value (e.g. one hundred elements). When the array is full, do not increase the size by just one element. The overhead of the resizing would be enormous. Instead, increase the size by e.g. a thousand elements or use a quotient ranging from 1.5 to 2 (better solution).

If STL is used, your implementation does not have to care about the allocation. Caution: only `vector` container is available in this homework, the other containers are disabled.

The interface of the required classes and some tests are included in the attached archive.

Update 2021-03-17: A solution of this problem may be used for code review if it passes all mandatory and optional tests with 100 % results.

Update 2021-03-18: The attached example tests were updated. Some additional checks were done in the basic test, however, these checks were not included in the attached source. The updated source lists them.

Sample data:		Download
Submit:	Choose File No file selected	Submit

Reference

- **Evaluator: computer**
  - Program compiled
  - **0** Test 'Zakladni test s parametry podle ukazky': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.000 s (limit: 5.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - **0** Test 'Test meznich hodnot': success
    - result: 100.00 %, required: 100.00 %
    - Total run time: 0.081 s (limit: 5.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - **0** Test 'Test nahodnymi vstupy (Add, List, Count)': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 0.006 s (limit: 4.919 s)
    - Mandatory test success, evaluation: 100.00 %
  - **0** Test 'Test nahodnymi vstupy (Add, Del, List, Count)': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 0.006 s (limit: 4.913 s)
    - Mandatory test success, evaluation: 100.00 %
  - **0** Test 'Test rychlosti (nahodne hodnoty)': success
    - result: 100.00 %, required: 50.00 %
    - Total run time: 0.625 s (limit: 5.000 s)
    - Mandatory test success, evaluation: 100.00 %
  - **0** Test 'Test nahodnymi hodnotami + test prace s pameti': success
    - result: 100.00 %, required: 25.00 %
    - Total run time: 0.029 s (limit: 5.000 s)
    - Optional test success, evaluation: 100.00 %
  - **0** Test 'Test rychlosti (fixni jmena, nahodna prijmeni)': success
    - result: 100.00 %, required: 75.00 %
    - Total run time: 0.562 s (limit: 5.000 s)
    - Optional test success, evaluation: 100.00 %
  - **0** Test 'Test rychlosti (nahodna jmena, fixni prijmeni)': success
    - result: 100.00 %, required: 75.00 %
    - Total run time: 0.458 s (limit: 4.438 s)
    - Optional test success, evaluation: 100.00 %
  - Overall ratio: 100.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00)
- Total percent: 100.00 %
- Early submission bonus: 0.50
- Total points: 1.00 * (5.00 + 0.50) = 5.50

SW metrics:		Total	Average	Maximum	Function name
	Functions:	**30**	–	–	–
	Lines of code:	**186**	**6.20 ± 5.30**	**27**	`CRegister::AddCar`
	Cyclomatic complexity:	**50**	**1.67 ± 1.22**	**6**	`CRegister::Transfer`