Land register

Submission deadline:	**2019-03-31 23:59:59**
Late submission with malus:	**2019-06-30 23:59:59** (Late submission malus: 100.0000 %)
Evaluation:	**6.6000**
Max. assessment:	**5.0000** (Without bonus points)
Submissions:	18 / 20 Free retries + 20 Penalized retries (-2 % penalty each retry)
Advices:	1 / 2 Advices for free + 2 Advices with a penalty (-10 % penalty each advice)

Your task is to implement class `CLandRegister`, which implements a simple register of land owners.

Our simplified land register maintains a list of land lots and their owners. Each land lot is described using 4 parameters: a region (string), the ID of the land lot ID (unsigned integer), city (string), and address (string). A land lot is identified either by pair (region, ID), or by pair (city, addr). For example a land lot has `region=Dejvice` and `ID=12345`. The pair positively identifies the land lot. On the other hand, `region=Dejvice` alone is not enough (there may exist many land lots with different IDs in Dejvice), similarly, `ID=12345` alone is not unique (there may exist land lots with this IDs in different regions). The same applies to pairs `city, addr`. A pair `city, addr` identifies the land lot positively. However, `city` alone and `addr` alone do not. We use **case sensitive** comparison when comparing strings `region`, `city`, and `addr`.

The owner is identified by string `owner`. One owner may own many land lots, one land lot must be owned by exactly one owner. Newly created land lots are owned by the state, the owner is set to an empty string. We use **case insensitive** comparison when comparing owner names.

The task is to implement class `CLandRegister` that maintains list of land lots, owners, and their relations. Moreover, the interface provides methods to update the ownership information in the register. The interface is:

Default constructor
: Creates an empty instance of the database (no land lots, no owners).

Destructor
: Releases all resources used by the instance.

`Add(city, addr, region, id)`
: the method adds a new land lot to the register. Parameters `(city, addr)` and `(region, id)` represent the land lot identification. The method returns `true` if the land lot was created, or `false` if the operation failed (the database already contains a land lot with the same `(city,addr)` pair or with the same `(region,id)` pair). The owner of the newly created land lot is the state (an empty string in our implementation).

`Del (city, addr) / Del (region,id)`
: the method removes a land lot from the register. The parameters positively identify the land lot either as a city and address (first variant) or as region and land lot ID (second variant). The method returns `true` if the land was deleted. If the corresponding land lot was not found, the method returns `false`. Note that the ownership does not play any role - the method may be used to expropriate a legitimate owner.

`GetOwner (city, addr, owner) / GetOwner (region, id, owner)`
: the methods retrieve ownership of a land lot. City/address or region/ID may be used to identify the land lot in question. Parameter `owner` is an output parameter, the method sets the value to the owner information from the register. Return value is `true` to indicate a success (output parameter must be filled in this case), or `false` to indicate a problem (the land lot was not found in the register). If the method fails, it must not modify the output parameters in any way. Even though we do not distinguish lower and upper case letters in owner names, the method must return owner name in the form it was entered. In other words, the register cannot simply convert all owner names to a fixed lower/upper case, instead, it must maintain the original strings.

`NewOwner (city, addr, owner) / NewOwner (region, id, owner)`
: the method sets new owner `owner` for the land lot identified by the parameters. The previous owner information is overwritten. City/address or region/ID may be used to identify the land lot to modify. Return value is `true` to indicate a success, or `false` to indicate a problem (the land lot was not found in the register, the owner already owns the land lot). If the method fails, the register is not modified in any way.

`Count (owner)`
: counts the number of land lots owned by the owner from the parameter. The result is a non-negative integer, zero if the `owner` does not own any land lot.

`ListByAddr ()`
: returns an iterator object (see below), to iterate through the list of all land lots in the register. The returned iterator must list all land lots sorted in an ascending order, the sort key is the name of the city and (if the city is the same) the address.

`ListByOwner (owner)`

returns an iterator object see (below), to iterate through the list of land lots owned by the owner from the parameter. The iterator must list the land lots in the order the owner bought them (i.e., in the order the owner registered them in our register). If the owner does not own any land lots, the result is an empty iterator (the iterator will indicate `AtEnd()` from the very beginning).

copy constructor, operator =
: the testing environment does not copy instances of `CLandRegister`, thus there is no need to implement them. You may decide to disable them (using C++11 notation `=delete`).

Class `CIterator` encapsulates a list of land lots and provides an interface to access the properties of individual land lots from the list. There is one land lot selected for access in the list, the properties of the selected land lot may be retrieved. Moreover, there is an interface to step forward to the next land lot in the list and an interface to detect the end of the list. The class is an example forward iterator (the simplest of iterators). The interface is:

`AtEnd`
: the method tests whether the end of the list was reached, or not. There is no land lot available once it returns `true`. On the other hand, if it returns `false`, then the land lot properties may be examined and we may step to the next record in the list. The iterator is designed to be used in a `while` loop. The condition would be `while ( ! it . AtEnd () )`, loop body then processes the properties of the land lot and finally, it invokes `it . Next()` to step to the next land lot in the list.

`Next`
: the method shifts the iterator to the next land lot in the list.

`City`
: retrieve the city from the selected land lot.

`Addr`
: retrieve the address from the selected land lot.

`Region`
: retrieve the region from the selected land lot.

`ID`
: retrieve the ID from the selected land lot.

`Owner`
: retrieve the owner of the selected land lot.

constructor, destructor, copy constructor
: the testing environment does not copy iterators. It just scans the returned list of land lots. Iterator instances are created by your implementation, the instances are returned from `ListByAddr / ListByOwner`. The return values may be copied or moved by means of copy/move constructors, the exact details depend on the actual implementations. A general recommendation is not to store any dynamically allocated structures in the iterator. The iterator may be implemented by means of two scalars (an array index and a reference to `CLandRegister`). In such case, the automatically generated copy/move constructors are just fine.

modifications
: the testing environment does not modify `CLandRegister` when it uses instances of `CIterator`. It first completes the iteration in the iterator (reaches `AtEnd ()`) and then it modifies `CLandRegister`. Therefore, the iterator does not have to copy the contents of `CLandRegister`. Instead, your `CIterator` may reference the `CLandRegister` instance and share its data.

Submit a source file with your implementation of `CLandRegister` and `CIterator`. The classes must follow the public interface below. 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). Moreover, you may add your auxiliary classes to the submitted file. The submitted file must include both declarations as well as implementation of the class (the methods may be implemented inline but do not have to). Do not add unnecessary code to the submitted file. In particular, if the file contains `main`, your tests, or any #include definitions, please, keep them in a conditional compile block. Use the attached source as a basis for your implementation. If the preprocessor definitions are preserved, the file maybe submitted to Progtest.

The interface of the class requires several methods that only differ in the way the land lots are identified. Spend some time with the design of the class. Do not just copy everything twice in your implementation (for example use some common private methods).

Although the memory limit is not very strict, there are some requirements on time efficiency. A simple linear-time solution will not succeed (it takes more than 5 minutes for the test data). You may assume `Add` and `Del` calls are rare compared to the other methods. Methods `NewOwner` and `GetOwner` are called very often, thus they must be very efficient (e.g. logarithmic time or amortized constant time). Next, the handling of iterators must be reasonably efficient.

Methods `Count` and `ListByOwner` are not called often in the mandatory and optional tests, thus their implementation does not have to be very efficient (a reasonable implementation of linear or n log n algorithm is enough). On the other hand, it must be better than linear in the bonus test. If not interested in the bonus test, focus on `NewOwner` and `GetOwner` at the cost of inefficient `Count`.

There are several ways to complete the bonus test. You may assume that the number of land owners is usually much smaller than the number of land lots.