Task 6. Final Project

The final project is a big task. To solve it you will need all the knowledge and skills of writing C++ programs acquired during the course.

Prerequisites, Goals, and Outcomes

Prerequisites: The students should have mastered the following prerequisite skills:

- writing console applications that read and write data using streams, including file streams;
- defining custom datatypes using classes; dealing with constructors, destructors, setters/getters, encapsulation, type aliasing and static methods;
- dealing with dynamic memory, properly allocating and deallocating memory resources, handling and sharing objects via pointers;
- properly managing resourses owned by custom classes.

Goals: This assignment is designed to reinforce the student's skills on solving complex programming problems using various features of C++.

Outcomes: The students successfully completing this assignment wI'll be able to design and develop C++ programs for tackling complex data analysis problems.

Description

There is a dataset in the form of a CSV text file. It contains a set of records that have some attributes. You need to read the data from a file and represent them as an object model. Further, the program will address this model with various analytical queries that need to be processed and return correct data.

The subject area, for which it is necessary to solve the indicated problem, refers to the employees of a certain organization. The dataset stores information about each employee as a record including the following attributes:

- employee name;
- employee age;
- employee department;
- employee position;
- boss name (a boss is also an employee from the same dataset);
- working days list.

Queries that can be executed against the object model with such data are the following:

- list of all employees;
- list of all employees whose age belongs to a given range;
- list of employees distributed by departments;
- fast lookup of an employee by their name;
- list of all subordinates of a given employee;
- list of employees working on given days.

You will be given with examples of CSV files (`employees.txt`, `employees2.txt`), so you will be able to test your solution on real data.

One of the key tasks to be solved within the framework of this project is the correct and efficient loading of initial data (or, as it is sometimes called, deserialization) from a file into computer memory, converting them into a form that is convenient for use.

There are a number of objectives to be solved on how exactly to store data so that they can be quickly accessed when performing different queries.

One of the first requirements you will be asked to implement is to develop a custom class to represent one *record* from a file, corresponding to a single employee. You will need to provide a convenient programming interface for initializing such a record object and accessing its attributes.

Another custom class that you will need to develop is an actual record storage, or *register*, which will provide a convenient programming interface for loading a dataset from a file (or more generally, from a read stream) and then manipulating it.

In order to solve this problem for any correct set of input data, you need to carefully decompose the problem so that each participant in the process could perform an assigned subtask clearly and correctly.

One of the key features of the data storage under development is the support of so-called *indexes* for fast retrieval of data projections required by a query. To do this, the class *register* has to organize, in addition to the main data store, also *index stores*, implemented using *maps* and *sets*. For example, a map-based index matches a *record* to a *key*, which is one of the attributes of the records and according to which the indexing is performed.

In order to be able to efficiently refer to the same record from different repositories and to ensure that one record is represented by strictly one object, the employee records have to be created *dynamically* in heap memory. Each employee record is stored by only its pointers to such a record in a store.

The downside of the flexibility that we get from working with records through pointers to them is the need for precise and accurate manual control of the lifetime of such records. To do this, you will have to provide a destructor in the register class, as well as methods for clearing all internal indices when reloading data.

In addition, if you want to make your *register* a correct copyable object, you will have to solve the shallow copy problem, and for this to correctly implement a *copy constructor* and a *copy assignment operation* for the register class.

When making queries to the register, selected data are presented in the form of some collection, which can then be printed on the screen (or output to a stream).

Project structure

You have to organize your program code as a valid `CMake` project. Consider proper module decompositions. Do not use any external libraries except the STL.

Tasks

Record class

In order to represent an individual record you have to create a class, whose fields reflect the record attributes mentioned above. The datatype of the fields have to reflect their nature properly. Employee name, department, position, boss name are string fields. Employee name is a numeric type. The working days must be represented as a collection of days, e.g. a vector of a set. Each day can be represented as a string or a value of enum type.

It is better to encapsulate the fields in the class implementation making them private. So in order to get access to them you have to consider proper getter methods (and setters if needed).

Each record representing an employee is a string line read from a CSV file. So it would be convenient to create a constructor that obtains a string line at its input, parses the line and fills the fields with a corresponding value. In addition (or alernatively) you may create a static factory method that constructs a new record object by its string representation. Note, that you have to store records by their pointers, so such a static method has to create a new record object in heap memory and return a pointer to it.

A string representing a single record consists of a sequence of attribute values separated by tab characters (`'t'`). The attributes follow each other in the order mentioned previously. The *Employee name*, *age*, *department* and *position* are not empty, so they have a corresponding string representation in a line. The *boss name* attribute can be empty for some records. In this case there are two consequent `'\t'` given in a row. The last attribute, *working days*, is represented by a set of 3-letter day abbreviations, e.g. "Mon", "Tue" etc. There can be from 1 to 7 of them separated by a tab character. The number of days is not known in advance, although it is guaranteed that there is at least one day present in a record.

An example of a record string is as follows:

```
Caius Mueller→20→acc→fellow→Lila Haigh→Tue→Thu
```

So, this is an employee named Caius Mueller, age 20, working in “acc” (accounting) department at a “fellow” position, and his boss is Lila Haigh. The working days are Tuesday and Thursday.

Note, that we use `→` character here to represent a tab character, however in fact normally it does not have any printable representation, so we use it purely for the sake of explanation.

Outputting a record to a stream

When mastering a program you will output employee records several times in a stream. To automate this operation you are requested to overload the `operator<<` in the record class for outputting to a given stream all the attributes (possibly, except for the working days, so that a record could fit a single line) decorated with their tag (e.g. “Name: Caius Mueller, age 20, ….”).

Register class

The register class aims to load, store, process and delete records properly.

Loading records from a file

The register class has to provide a way to load data from a file by its name. You may create a separate method that obtains a path to a CSV file and loads data from it to a register. To do this, you may read the file line by line, and pass each line to a method of the record class, which creates a record object from a string line and returns a pointer to it. Further, such a pointer is added to a local storage and manages properly. Alternatively, you may implement a similar method in the register class, that is, to pass a record line, parse it properly, create a new record object in heap memory and store a pointer to it.

We suggest to consider the approaches discussed in the “I/O and Streams” and “Strings and Streams” sections for reading a file.

Storing employees by their pointers

There must be exactly one employee object for a single loaded record. Firstly, you have to select a storage for employee objects. For example, it can be a vector container. The point is that you have to avoid inefficient copying of employee objects stored in such a storage. Note, that the containers of the standard library manipulate their elements using *value semantics*, which means that each time when a container reallocates its memory it copies elements themselves. In case of such heavy elements as record class objects, such copying is inefficient and does not give you a stable handle for an individual object, say a pointer to a record. Indeed, occasionally a vector of records reallocates its internal buffer with the previous copies of records being removed and new ones being added.

Thus, storing record objects in a vector class is a bad idea. Instead, you have to consider storing *pointers to records*. This way, copying of pointers is cheap and you may have a lot of copies of the pointer to a particular record, which are stored in a number of containers at once. So, the type of a vector storing records by pointers can look as follows:

```
typedef std::vector<Employee*> EmplVector;
```

where `Employee` is a type name for a record class.

In order to have stable pointers, namely, the pointers to record objects that are present in memory persistently, you have to manage their creation manually. That is, you have to create them in heap memory using the `new` operator. It can be done in a parsing method (static one) of the register class, as it was discussed previously.

All the objects that are created manually in heap memory, must be removed manually as well later, using the `delete` operator. The good place to do so, according to the RAII idiom, is a destructor of the register class. So, you have to iterate over the collection of record pointers and delete each object properly. Note, that it is not enough to simply delete the pointers themselves, e.g., by calling the `clear()` method of a vector. In this case, the objects these pointers point to remain undeleted.

Indexing records for performing fast retrieval

Requests performed for a loaded dataset require a lot of lookups for stored records. Each request requires to consider each record to find necessary information. It takes a lot of time since iterating all records request linear time. However, some requests may consider a specific order of keys they look for. Thus, using an ordered collection represented by a set or a map can reduce the time needed for searching a specific key and set it to a logariphmical dependence of the number of records.

The approach you have to implement in you project considers creating an ordered collection for each attribute you have to look up during the performed requests. We call such an auxilliary collection an *index*. Indexes can be created after loading all records from a file, e.g. at the first addressing of a corresponding attribute, or together with loading data from the file after adding next record to the main storage.

We consider the main storage of the register and index storages equal in the sense of their dealing with record objects — they all consider records through pointers. However, only the main storage use its pointers to delete objects when the time comes. Indexes only store the record pointers, considering them valid during their lifetime.

We are considering individual indexes when discussing individual requests below.

Clearing records

It is necessary to provide an ability to remove all the loaded records from the storage upon a user's request. Implement a method clearing the records properly: one should delete all allocated memory and then clear the main storage and index storages from invalid pointers. Note that this operation correlates to freeing memory done in the register destructor, so in order to avoid code duplication consider a common procedure for both cases.

Copying register

Since the register class manages dynamically created records through their pointers, the default implementation of a copy constructor will not work properly, because it simply copies pointers (*shallow copy*). Proper copying requires creating a copy of each record (check whether it can be done by a default implementation of the copy constructor of the record class), adding pointers to the new records and re-indexing them in a new storage. To do this, you have to overload the copy constructor for the register class. Also, you have to overload the copy assignment operator. It can be done using copy-and-swap approach.

Requests to a dataset

Getting a list of all employees

This request requires getting access to the underlying storage, e.g. to a vector of employee records. You can expose it to the outside of the class using a getter method. However, make sure that this collection is protected from changing and also you have to avoid copying it. So, make such a getter const, for example:

```
const EmplVector& getStorage() const;
```

It can also be used for getting employees whose age belongs to a given range. In this case you simply iterate over the employees collection and consider only those records, whose age attribute belongs to the given range.

Lookup of an employee by their name

You have to provide a method that returns a (pointer to the) record, whose attribute name matches a given string. In order to find a corresponding record quickly, you have to create a map-based index considering employee names as keys and pointers to employee objects as values, e.g.

```
std::map<std::string, Employee*>
```

When indexing, you populate the map with data and then extract an employee by their method quickly.

Getting a list of employees distributed by departments

You have to create a map-based index considering department names as keys and a collection of employee objects as value, e.g.

```
std::map<std::string, std::vector<Employee*> >
```

Having such a collection created and populated with data, you may expose it outside with a getter method taking into account the issues related to constancy and avoiding unnecessary copying.

Getting a list of subordinates of a given employee

First, you have to provide a map-based index storing of “employee — collection of employees” using an approach similar to the one discussed in the previous section. Then you extract all the direct subordinates of a given employee by using a fast lookup in an index. In order to consider all the indirect subordinates, you have to *recursively* repeat the same action for all direct subordinates. All extracted subordinates have to be stored in a local collection (vector), which is returned by value.

It is guaranteed that there are no circular dependencies between employees.

Getting a list of employees working on given days

This task requires you to prepare a set employees, so each employee from the set works on at least one of the set of given days. A set of days is given as `std::set< Day >` object, where `Day` is a datatype choosen earlier for the day representation. You have to provide a getter which returns a set of employees, e.g.:

```
std::set<Employee*>
```

We use a set here, not a vector, in order to eleiminate any duplicates appearing when an employee works on more than one day from the given set of days.

Main program

You are requested to create a full-fledge console application that provides a UI to deal with the tasks implemented in your classes. An application may provide a simple menu with choices that can be choosen by a user, e.g., by providing a character. For example:

```
Employee database
=================

(L) Load a file
(C) Clear a dataset
(N) Print number of records in a dataset
(P) Print all records
(E) Print an employee by their name
(A) Print all employees with an age in a given range
...
(X) Exit

Choose an action: _
```

The program should provide an ability to repeat actions until a user decides to leave the program choosing a corresponding item from the menu.

Submission details

You have to submit a zip file containing only the source code of your application and a `CMakeLists.txt` file. Do not include any auto-generated files. If you use additional resources, include them into `/resource/` subfolder of your zip.