NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C05**

Movie Database I

The goal of this task is to implement the first part of a simple application that will allow us to simulate work with a database of movies. By this we mean the ability to store individual movies (i.e., create their instances and store them in a standard vector) and then query them (i.e., search for individual movies matching the provided search criteria and print them to the standard output).

We need to record the following data items for each movie: movie name (of type `std::string`), year of filming (`unsigned short`), genre (`std::string`), rating (`unsigned short`), and a set of names of actors (`std::string`) who played in a given movie. The set itself will be realized using another type of a standard container, namely `std::set`, which can be found in the `<set>` library. All data members are mandatory in general, but the number of actors can be completely arbitrary, even zero.

In order to represent the described movies, we will create a class named `Movie`, each specific movie will then be represented by one instance of this class. All the above-mentioned pieces of data will be implemented as private data members, access to them from the outside will only be possible via public parameterless methods `name`, `year`, `genre`, `rating`, and `actors`, all returning unmodifiable references to the relevant items (in the case of title, genre, and actors) or copies of the respective values (for year and rating). In addition, these methods must be declared as constant (so that we can call them even on non-modifiable movie instances), and for the sake of efficiency, we will program them as inline methods.

The `Movie` class will also offer the following two constructors. The first of them will save copies of the passed items, the second one will appropriate the items passed by rvalue references. It is expected that the mechanism of member initializer lists will be used to store the individual items in both cases.

- `Movie(const std::string& name, unsigned short year, const std::string& genre, unsigned short rating, const std::set<std::string>& actors)`
- `Movie(std::string&& name, unsigned short year, std::string&& genre, unsigned short rating, std::set<std::string>&& actors)`

The last method of the `Movie` class will be `void print_json(std::ostream& stream)`. Through it, we will be able to print a given movie to the provided output stream. It is required to make calling of this function possible even without its only parameter specified, in which case the standard output `std::cout` will be assumed. We will print the movies in the form of a JSON object, its structure must correspond to the following template:

```
{ name: "Bobule", year: 2008, genre: "comedy", rating: 65, actors: [ "Krystof Hadek", "Te
reza Voriskova" ] }
```

In addition to spaces and the overall structure, we must also observe the order and names of the individual items and enclose text values in double quotes. We will enumerate the names of the actors in exactly the same order as they will be returned to us by the relevant set container iterator. If a given movie does not have even a single actor, we will then not include the `actors` property at all. If the actors are present, we will, of course, not print the separator after the very last of them. There are no line breaks involved, not even at the end after the closing curly brace of the entire JSON object.

We implement the database itself using a single instance of the standard vector container `std::vector`, into which we will gradually insert individual movie instances. For this, we can use, for example, the `push_back` or better `emplace_back` method. However, we will also offer the users the possibility of importing the movies from input files or from general input streams. We will propose a `Database` class for this purpose, assuming that its declaration will be placed in a `Database.h` header file. It is not intended for this class to store the content of our database, it will only encapsulate the import functionality through the following two public static methods:

- `static void import(const std::string& filename, std::vector<Movie>& db)`: opens a given input file, imports the movies, and inserts their instances into the prepared vector
- `static void import(std::istream& strem, std::vector<Movie>& db)`: performs the import of movies from a given input stream and inserts their instances into the prepared vector

In both cases, we assume the input data to be in the CSV format. This means that there will be one movie on each individual line, and entirely empty lines will be skipped. For a given movie, we gradually expect its title, year, genre, rating, and names of actors. Individual fields will be separated by a semicolon, names of individual actors by a comma.

Movie title and genre fields must be non-empty strings. As for the allowed values of numeric fields, year of filming is expected to be from the closed interval 1900 to 2100, movie rating from 0 to 100. We can assume that the separators used will not appear anywhere inside the values of our items. Since we will add several more fields in the follow-up assignment, it is worth decomposing the code appropriately. Let us also remember that the number of actors can be zero, as can be seen in the second movie within the following example:

```
Dira u Hanusovic;2014;comedy;49;Tatiana Vilhelmova,Ivan Trojan,Klara Meliskova
Vlastnici;2019;comedy;74;
```

Use of the global function std::getline is expected for the input parsing, this time also in the variant with an optional third parameter, via which a separator other than the default one can be specified. Let us note that a string can easily be transformed into a stream. In particular, using std::istringstream from the <sstream> library. We will insert movie instances into the database container as efficiently as possible, i.e., using the already mentioned emplace_back method in combination with the std::move construct.

We will handle all error situations using structured exceptions, in the form of a structure struct Exception { int code; std::string text; }, where the first item will contain the numeric code of the error type and the second one a text string explaining the cause of the error in more detail. Specifically, we expect the following behavior:

- Code 1 (input errors)
  - Unable to open input file <filename>: it is not possible to open a given input file
- Code 2 (parsing errors)
  - Missing field <name> on line <line>: the corresponding field is missing, i.e., there is no other field in the input string
  - Empty string in field <name> on line <line>: empty string for a text field that does not permit empty values
  - Invalid integer in field <name> on line <line>: invalid numeric value for number fields as a response to the std::invalid_argument exception from the function std::stoi
  - Overflow integer in field <name> on line <line>: analogously, overflowed numeric value as a response to the exception std::out_of_range
  - Malformed integer in field <name> on line <line>: analogously, not well formed numeric value as a response to a failed position test
  - Integer out of range <min, max> in field <name> on line <line>: numeric value out of the range of permitted values

Parameters in angle brackets will, of course, be replaced by particular values, the brackets themselves will be preserved: filename is a name of the required file, name is a name of the problematic field (i.e., name, year, genre, rating, or actors), line is a line number in the input file (counting from 1), and min and max is the interval of the allowed values for numeric fields.

If we encounter any problems when parsing a particular movie record, we will not instantiate that movie, we will throw an appropriate exception, and we will not continue processing anything remaining in the input. All movies that we have already managed to insert into the database (vector of movies) will remain in it, untouched.

Finally, we will implement two global functions with which we will simulate the evaluation of queries over our database. In both cases, we will simply iterate through all the movies in the passed container and print the ones we were looking for in the intended form into the specified output stream. Each such movie found will also be terminated by the end of line. Both the functions will be declared in a header file Queries.h.

- void db_query_1(const std::vector<Movie>& db, std::ostream& stream = std::cout):
  we will find all movies (i.e., no filtering is performed); we will print each of them as a complete JSON object (using the function we prepared)

- `void db_query_2(const std::vector<Movie>& db, std::ostream& stream = std::cout)`: we will find all comedies (*comedy*) filmed before the year *2010*, in which *Ivan Trojan* or *Tereza Voriskova* played; for each of them, we will print only the corresponding movie titles

Similarly to the previous assignment, the course of the entire test will again be controlled directly from the `main` function. I.e., you will not submit it, since its already prepared variant will be used instead. It first creates an instance of our database, i.e., the vector for movies. Subsequently, it will manually create and insert new instances of movies, import them from input files, or call the discussed query functions, all that repeatedly and in any order.

Divide all your code into individual modules with header files. The class `Database` and functions `db_query_*` must be declared in header files `Database.h` and `Queries.h`, respectively. Submit all the created source files (`*.cpp` and `*.h`) except the main file `Main.cpp`.

The specific goal of the task is to verify the ability to work with the following constructs: design and use of ordinary data classes, design of their parameterized constructors in combination with member initializer lists, design of inline functions, use of the `std::move` construct and work with rvalue references, getting acquainted with the `std::set` container and advanced ways of inserting elements into the `std::vector` container using the general `emplace` mechanism, and, finally, working with the `std::getline` function with a separator different to the default one, combined with `std::istringstream` streams. In other words, it is necessary to use all these constructs within your solution.

We assume compliance with all the procedures we have already learned, as well as the requirements we have for the tasks. Specifically, do not forget named constants for the semicolon and comma delimiters we use in the input CSV files (so that they can easily be changed if need be), as well as using the convention of postfixing private data member names in our classes with the `_` symbol.

Each local variable is supposed to be declared in the most specific block, i.e., only in such a block (which is always defined by a pair of curly braces `{}`) where we really need to use it. Any other auxiliary methods for parsing movies should again be added as private static member functions of the `Database` class, not as separate global functions. Inline functions must be programmed in a way that not only their declarations but also their full definitions are known at the places of use. In our case, this means that we have to implement their bodies directly in the given header file.

When parsing individual fields of a particular movie from the input, it would not be appropriate to use a loop (either `while` or `for`) since we need to process each field differently. In other words, loops are generally only useful in situations where each iteration proceeds, let us say, similarly, or at least has some sufficiently significant common ground. In our case, however, we would have to re-branch the body of the loop for each individual case (perhaps using the `switch` construct or otherwise) and solve it separately. Regardless of the obvious inefficiency, such a solution would primarily be a bad design, and so we will avoid it.