NPRG041 – 2023/24 Winter – Labs MS – **Small Assignment C09**

Movie Database III

Within the last task belonging to the topic of our movie database, we will start with the existing application and extend it by adding several auxiliary indices and especially new queries. The purpose of these indices, as auxiliary data structures based on various standard containers, will be to simulate traditional database indices, and thus enable more efficient evaluation of our queries.

We will specifically create three of the mentioned indices: we will call them the index of titles, years, and actors. They will be implemented using the standard containers `std::map`, `std::multimap`, and `std::unordered_multimap`, respectively, in the specified order. In all cases, we assume that an empty instance of a given index will first be prepared in the `main` function, our task will be to implement the following global functions through which we will be able to fill a given index with the corresponding records based on the current state of the database. In other words, we will appropriately index every title that will exist in the database container at a given moment.

- `void db_index_titles(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `std::map<std::string, std::shared_ptr<Title>>& index`
  `)`: index allowing to search for titles based on their (unique) names
- `void db_index_years(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `std::multimap<unsigned short, std::shared_ptr<Title>>& index`
  `)`: index for searching titles based on years they were filmed; we assume that `std::less<unsigned short>` will be used as the comparison functor when creating the container for this index; it already exists, we will therefore not implement it, of course
- `void db_index_actors(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `std::unordered_multimap<Actor, std::shared_ptr<Title>>& index`
  `)`: index enabling to find titles by actors who played in them; when creating the container for this index, `std::hash<Actor>` will be used as the hash functor, and, analogously, `std::equal_to<Actor>` will be used as the comparison functor; the first mentioned one does not exist, and we will therefore need to implement it by ourselves (in a header file) as a specialization of the hash functor template, i.e., in the form of a structure `template<> struct std::hash<Actor> { … }`, within which we implement just a single method, namely the parentheses operator `()` in the form of a member function `std::size_t operator()(const Actor& actor) const noexcept`, where we just return a hashed value based on the actor surname, for which we will use the existing functor `std::hash<std::string>`; as for the comparison functor `std::equal_to<Actor>`, it suffices to implement the equality comparison operator `==` in the form of a global function, i.e., in the form of `bool operator==(const Actor& actor1, const Actor& actor2)`

We will preserve all four existing database queries unchanged and add the following new queries in the same style. However, as for the interface point of view, we will no longer pass them a reference to the entire database, but only a relevant index discussed above. It continues to apply that for each title found, we print the result in the required format to a specified output stream. Alternatively, we will print a message that no suitable record could be found. We again terminate the printing of each record with the end of line.

- `void db_query_5(`
  `const std::map<std::string, std::shared_ptr<Title>>& index,`
  `const std::string& name,`
  `std::ostream& stream = std::cout`
  `)`: based on the title index, we find a specific title that has a name `name`; if we find it, we print it in the form `"name" -> title`, where `name` will be replaced by the value of the intended name and `title` will be replaced by a complete JSON object for a given title; if the searched title does not exist, we will only print `"name" -> Not found!`, where `name` will again be replaced by the intended name

- `void db_query_6(`
  `const std::multimap<unsigned short, std::shared_ptr<Title>>& index,`
  `unsigned short year,`
  `std::ostream& stream = std::cout`
  `)`: using the year index, we find all titles that were filmed in a year `year`; we print each such title in the form `year -> "name"`, where `year` will be replaced with the value of the year we are looking for and `name` with the name of a title found; if there is no suitable title, we will analogously print `year -> Not found!` after the replacement

- `void db_query_7(`
  `const std::multimap<unsigned short, std::shared_ptr<Title>>& index,`
  `unsigned short begin, unsigned short end,`
  `std::ostream& stream = std::cout`
  `)`: using the same index again, we find all titles that were filmed in a year belonging to the right-open interval [`begin`, `end`); we print the found titles in the same format as in the previous query; if we do not find any, we print `[begin, end) -> Not found!` after the replacement

Finally, we will add the following new queries. In their case, however, we will no longer be printing anything to the output, since we will always pass the found or calculated result to the caller in the form of a return value. The intended standard algorithms can be found in the `<algorithm>` library.

- `std::vector<Title*> db_query_8(`
  `const std::unordered_multimap<Actor, std::shared_ptr<Title>>& index,`
  `const Actor& actor`
  `)`: based on the actor index, we find all titles in which a given actor `actor` played; the result will be returned in the form of a vector of C-style pointers to the corresponding titles

- `std::vector<std::shared_ptr<Title>> db_query_9(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `const Actor& actor`
  `)`: the goal is again to find the titles in which a given actor `actor` played; this time, however, we will achieve this by first copying all the titles from the database (meaning the shared smart pointers to them) into an empty output vector container we created, using the function `std::copy`; subsequently, using the `std::remove_if` function, we remove those titles that do not suit us, using a predicate in the form of a functor implementing a method `bool operator()(const std::shared_ptr<Title>& title_ptr)`; finally, we will order all the remaining titles, i.e., those we searched for, primarily in descending order by years of their filming and secondarily in ascending order by their names, using a functor implementing a method simulating the behavior of the `<` operator, specifically in the form of a method `bool operator()(const std::shared_ptr<Title>& title_ptr_1, const std::shared_ptr<Title>& title_ptr_2)`

- `std::vector<std::shared_ptr<Title>> db_query_10(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `const std::string& genre`
  `)`: we find all titles having a genre `genre` by copying the matching titles into an empty output vector container we created, using the `std::copy_if` function and a predicate implemented using a lambda expression; finally, we sort the titles, again using a lambda expression simulating the behavior of the `<` operator, assuming we want to sort the titles in ascending order according to their names

- `int db_query_11(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `Type type, const std::string& genre`
  `)`: the goal is to calculate the integer average rating of titles having our enumeration type `type` and genre `genre`; for the processing of individual titles, we will use the function `std::for_each`, processing each particular title using our own functor implementing a method `void operator()(const std::shared_ptr<Title>& title_ptr)`

- `int db_query_12(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `const std::string& genre`
  `)`: we calculate the overall number of titles having a genre `genre`; we will again use the function `std::for_each`, but this time, we will solve the processing of a particular title using a lambda expression

Submit all source files (`*.cpp` and `*.h`) your created except for the main file `Main.cpp`. Again, we assume `#include` directives for the header files `Database.h` and `Queries.h`. Of course, compliance with the usual requirements for our tasks is expected. If the use of certain constructs or functions has been prescribed within individual queries, it is necessary to abide by such an intention.

The objective of this task is to verify the capability of working with standard containers `std::map`, `std::multimap`, and `std::unordered_multimap` with our own classes, structure `std::pair`, predefined functors `std::less`, `std::hash`, and `std::equal_to`, standard algorithms over containers, specifically functions `std::copy`, `std::copy_if`, `std::remove_if`, `std::sort`, and `std::for_each`, as well as functors and lambda expressions in general.