NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C09**

Movie Database III

Within the third assignment 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 five of the mentioned indices: let us call them the index of titles by names, actors by years, titles by years, titles by actors, and cast by genres. They will be implemented using the standard containers `std::map`, `std::multimap`, and `std::unordered_multimap` (from libraries `<map>` and `<unordered_map>`). 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 content of the database.

- `void db_index_titles_by_names(`
  `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_actors_by_years(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `std::map<unsigned short, std::set<Actor>>& index`
  `)`: index for finding actors by their years of birth; when inserting records into this index, operator `[]` on the outer map is expected to be used

- `void db_index_titles_by_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

- `void db_index_titles_by_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 a hash functor, and, analogously, `std::equal_to<Actor>` will be used as a 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 `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 an instance of 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& actor_1, const Actor& actor_2)`

- `void db_index_cast_by_genres(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `std::multimap<`
  `std::tuple<std::string, unsigned short>,`
  `std::tuple<std::string, std::string, std::shared_ptr<Title>>`
  `>& index`
  `)`: this index will allow storing the cast information in titles based on their genres and years, specifically in the form of triples (actor first name, actor surname, pointer to a given title) based on pairs (title

genre, year of title filming); to represent these pairs and triples, we will use the `std::tuple` structure from the `<tuple>` library

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 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 index of titles by names, 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 title name

- `void db_query_6(`
  `const std::map<unsigned short, std::set<Actor>>& index,`
  `unsigned short begin, unsigned short end,`
  `std::ostream& stream = std::cout`
  `)`: based on the index of actors by years, we calculate the overall number of actors born during the years belonging to the right-open interval [`begin`, `end`); the result will be printed in the form `[`*begin*`, `*end*`) -> `*count* `actors` or `actor`, depending on the determined *count*.

- `void db_query_7(`
  `const std::multimap<unsigned short, std::shared_ptr<Title>>& index,`
  `unsigned short year,`
  `std::ostream& stream = std::cout`
  `)`: using the index of titles by years, 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_8(`
  `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

We further implement a query that enables finding titles based on a general search condition implemented in the form of a separate function.

- `void db_query_9(`
  `const std::vector<std::shared_ptr<Title>>& db,`
  `const std::function<bool(const Title*)>& predicate,`
  `std::ostream& stream = std::cout`
  `)`: finds all titles that satisfy a selection condition evaluated by the provided `predicate` function; this function expects a single parameter in the form of a non-modifying C-style observer pointer to a title for which the condition is to be evaluated, while it returns `true` if and only if the given title should be included in the query result; to allow the use of not only ordinary functions but also functors or even lambda expressions later on for these conditions, we will pass this parameter using the `std::function` structure from the `<functional>` library; each found title will be printed in the format `"`*name*`"`, where *name* is to be replaced with the title name; if no title is found, we simply output `Not found!`

For the previous query, we also prepare the following two functions with specific conditions. The first one will be implemented as an ordinary global function, the second as a functor.

- Global function `bool predicate_Q9_movies(const Title* title)`: finds all movies in which at least 3 actors performed
- Functor `Predicate_Q9_Titles` implementing the `()` operator as a member function `bool operator() (const Title* title) const`: finds all titles with a rating of at least 80 where `Tatiana Vilhelmova 1978` played

Finally, we will add the following new queries. In their case, however, we will no longer be printing anything to the output. We will always pass the found or calculated result to the caller in the form of a return value instead.

- ```
  std::vector<Title*> db_query_10(
    const std::unordered_multimap<Actor, std::shared_ptr<Title>>& index,
    const std::string& surname
  ```
  ): based on the index of titles by actors, we find all titles in which an actor with surname `surname` played; the result will be returned in the form of a vector of traditional C-style observer pointers to the corresponding titles; when iterating over individual records in the index, it is expected that structured binding `auto&& [key, value]` will be used
- ```
  std::vector<std::string> db_query_11(
    const std::multimap<
      std::tuple<std::string, unsigned short>,
      std::tuple<std::string, std::string, std::shared_ptr<Title>>
    >& index,
    const std::string& genre, unsigned short year
  ```
  ): using the last defined index cast by genres, it finds names of actors and names of titles in titles with genre `genre` filmed in year `year`; each found record will be returned as an `std::string` containing a JSON object in the following format: `{ name: "`*name*`", surname: "`*surname*`", title: "`*title*`" }`, where *name* is the actor first name, *surname* their last name, and *title* name of a given title

During the work on the assignment, you will use the current version of your complete movie database project. However, you will only submit the module for database queries themselves. Specifically, you will most likely only submit the header file `Queries.h` and the corresponding source file `Queries.cpp`. No other files from the original project should be submitted. In other words, your queries will be evaluated against the implementation of the entire database contained in the prepared test, not your own.

The test itself will contain the `#include` directive for the header file `Queries.h` only. Within it, you may (in addition to the required standard libraries) include only the `Movie.h` header file, through which you will gain access to everything necessary, particularly everything related to titles, movies, series, and actors. In their case, it also means the implementation of the comparison operator `==` (i.e., also do not submit it).

You are expected to adhere to the usual requirements for our assignments. If specific constructs or functions were prescribed within individual queries, it is necessary to abide by such an intention. The objective of this task is to verify your ability to work with standard containers such as `std::map`, `std::multimap`, and `std::unordered_multimap` with our own classes, structures `std::pair`, `std::tuple`, and `std::function`, predefined functors `std::less`, `std::hash`, and `std::equal_to`, and functors as such in general.