NPRG041 – 2025/26 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.

Let us note right at the beginning that it is actually not necessary to have the previous assignment in this thematic series fully implemented for the successful completion of this task. The database implementation itself will not be submitted, only the newly added indices and queries. It will therefore suffice to familiarize yourself with the public interface of the `Title`, `Movie`, `Series`, and `Actor` classes only. This means the interface of their constructors, getter methods for data members, their printing using the `<<` operators, and distinguishing types of titles using the `Type` enumeration.

We 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. For their implementation, we will use standard containers `std::map`, `std::multimap`, and `std::unordered_multimap` (from libraries `<map>` and `<unordered_map>`), and for increasing clarity when working with them, we will also assume the following type aliases.

- `using db_index_titles_by_names_t = std::map<`
  `std::string, std::shared_ptr<Title>`
  `>`: index allowing to search for titles based on their (unique) names
- `using db_index_actors_by_years_t = std::map<`
  `int, std::set<Actor>`
  `>`: 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
- `using db_index_titles_by_years_t = std::multimap<`
  `int, std::shared_ptr<Title>`
  `>`: index for searching titles based on years they were filmed; we assume that `std::less<int>` will be used as the comparison functor when creating the container for this index; it already exists, we will therefore not implement it
- `using db_index_titles_by_actors_t = std::unordered_multimap<`
  `Actor, std::shared_ptr<Title>`
  `>`: 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., `bool operator==(const Actor& actor_1, const Actor& actor_2)`; we perform the actual comparison using the already implemented operator `==` on tuples `std::tuple` from the `<tuple>` library, which we artificially create for this purpose using the `std::tie` function
- `using db_index_cast_by_genres_t = std::multimap<`
  `std::tuple<std::string, int>,`
  `std::tuple<std::string, std::string, std::shared_ptr<Title>>`
  `>`: 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)

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 populate a given index with the corresponding records based on the current content of the database.

- `void db_index_1(const database_t& database, db_index_titles_by_names_t& index)`
- `void db_index_2(const database_t& database, db_index_actors_by_years_t& index)`
- `void db_index_3(const database_t& database, db_index_titles_by_years_t& index)`
- `void db_index_4(const database_t& database, db_index_titles_by_actors_t& index)`
- `void db_index_5(const database_t& database, db_index_cast_by_genres_t& index)`

We preserve all the five 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 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_6(`
  `const index_titles_by_names_t& index,`
  `std::string_view 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 its full JSON object; otherwise, we only print a message `Not found!`

- `void db_query_7(`
  `const index_actors_by_years_t& index,`
  `int begin, int 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 a given right-open interval [`begin`, `end`); the result will be printed in the form *count* `actors` or `actor`, depending on the determined *count* (e.g., `7 actors` or `1 actor`)

- `void db_query_8(`
  `const index_titles_by_years_t& index,`
  `int 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 as a string with a JSON object `{ name: "`*name*`", year: `*year*` }`, where *name* will be replaced with title name and *year* with its year of filming; if there is no suitable title, we just print a message `Not found!`

- `void db_query_9(`
  `const index_titles_by_years_t& index,`
  `int begin, int end,`
  `std::ostream& stream = std::cout`
  `)`: using the same index, we find all titles that were filmed in a year belonging to a 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 again only print `Not found!`

In the following two queries, we will do without indices, and so we will once again work with the database directly. In the first of them, we will try out working with a container that uses a sorting functor other than the default one. The second one will then allow finding of titles based on a general search condition implemented in the form of a separate function or functor.

- `void db_query_10(`
  `const database_t& database,`
  `int begin, int end,`
  `std::ostream& stream = std::cout`

): we find every single casting of actors born in the years belonging to an interval [`begin`, `end`); in other words, we go through all titles and find all matching actors within them; we are only interested in the actors themselves, all duplicates will be preserved; since we also want to print these actors in a specific order (different from the default one, i.e., the one we defined in the previous task), we will first store them internally in an auxiliary multiset container `std::multiset`, in which we will use our own functor for sorting; specifically, it will be a class `Comparator_Q10_Actors` with a public method `void operator()(const Actor& actor_1, const Actor& actor_2) const`, which will simulate the behavior of the `<` operator for the ascending order based on years of birth, first names, and last names; for its elegant implementation we will again use artificially created tuples `std::tuple`, this time obtained using the `std::make_tuple` function; for each found actor, we print its complete JSON object; if none exists, we print the `Not found!` message

- ```
  void db_query_11(
    const database_t& database,
    const std::function<bool(const Title*)>& predicate,
    std::ostream& stream = std::cout
  ```
  ): we find all titles that satisfy a general 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; for each found title, we print its whole JSON object; if no title is found, we print `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. In neither case should you create named constants for the given particular values, as it would go against the purpose of these functions.

- Global function `bool predicate_Q11_movies(const Title* title)`: we find all movies (not series, i.e., only titles with type `Type::MOVIE`) with at least three actors that are not comedies
- Functor `Predicate_Q11_Titles` implementing a method `bool operator()(const Title* title) const`: we find all titles with a rating of at least 80 where `Tatiana Vilhelmova 1978` played

Finally, we add the following two queries. We return back to working with indices, and we will no longer print anything to the output stream, because the found or calculated results will always be passed to the caller in the form of a return value instead.

- ```
  std::vector<Title*> db_query_12(
    const index_titles_by_actors_t& index,
    std::string_view surname
  ```
  ): based on the index of titles by actors, we find all titles in which an actor with a 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_13(
    const index_cast_by_genres_t& index,
    std::string_view genre, int year
  ```
  ): using the last defined index cast by genres, we find names of actors and names of titles in titles with a genre `genre` filmed in a 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

As already explained in the introduction, during the work on the assignment, you will use the current version of your movie database project (provided you have it). However, you will only submit the module for database queries as such. More precisely, you will only submit queries Q6 to Q13, all indices and related code. If you also submit older queries Q1 to Q5, nothing happens, they will just not be tested. Specifically,
  ```

you will most likely only submit the `Queries.h` header file and the corresponding `Queries.cpp` source file. 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 the one you created on 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 `Storage.h` header file, through which you will gain access to everything necessary, particularly everything related to titles, movies, series, and actors.

As part of this assignment, we are to implement, among other things, a specialization of the hash functor `std::hash` and the equality test operator `==` for our actors. Naturally, it would make the most sense to place both of them into the module of actors. However, since it will not be submitted, place them into the query module instead so that they can be tested.

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