NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C12**

Movie Database IV

The purpose of the last assignment in the movie database topic is to further extend the existing project by adding several new queries, within which we will use various standard algorithms (libraries `<algorithm>` and `<numeric>`), functors, lambda expressions, fake iterators (library `<iterator>`), as well as ranges and views (library `<ranges>`).

Since we already have implemented our custom flexible array container from the previous assignments, it would be a shame not to use it. In other words, it is expected that the current concept of our database represented as `std::vector<std::shared_ptr<Title>>` will be replaced by `lib::Array<std::shared_ptr<Title>>`. At least in case you really have the flexible array implemented and sufficiently debugged. In other words, this change is not compulsory.

We then add the following six new database queries to our project. All prescribed constructs and procedures need to be faithfully followed.

- `database_t db_query_14(`
  `const database_t& database,`
  `const Actor& actor`
  `)`: we find titles in which a given actor `actor` played; we achieve this by first copying all the titles from our database (meaning the shared smart pointers to them) using the `std::copy` function into an output container of our `database_t` type we prepared; 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 `Predicate_Q14_Actor`; finally, we order all the found titles, primarily in descending order by years of their filming and secondarily in ascending order by their names, using the `std::sort` function and a functor `Comparator_Q14_Years` simulating the behavior of the `<` operator

- `void db_query_15(`
  `const database_t& database,`
  `int year,`
  `std::ostream& stream = std::cout`
  `)`: we find all movies (titles with a type `Type::MOVIE`) filmed in a particular year `year` by copying this time only the matching titles into an empty auxiliary container of our type `database_t`; we will use the `std::copy_if` function for that, a predicate implemented using a lambda expression, and a fake iterator instance for the output obtained by calling `std::back_inserter`; we then 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; finally, we output all the titles in a transformed form to the provided output stream using the `std::transform` function; the actual transformation will again be performed via a lambda expression; for a given title, it returns a string `{ name: "`*title*`", year:` *year* `}`, where *title* and *year* are expected to be replaced by the title name and year of filming, respectively; as for the output, we will use a fake iterator `std::ostream_iterator<std::string>`, terminating each record by printing a line ending

- `std::optional<int> db_query_16(`
  `const database_t& database,`
  `Type type,`
  `std::string_view genre`
  `)`: we calculate the integer average rating of titles having our enumeration type `type` and a genre `genre`; for iterating over the individual titles, we will use the `std::for_each` function, processing each particular title using a functor `Visitor_Q16_Rating`; we implement it in a way that all the logic of calculating this average will be encapsulated within it; for a zero number of titles, we return `std::nullopt`, as expected; let us further note that the `std::for_each` function creates and subsequently uses its own copy of the provided functor, it then returns it as its return value

- `size_t db_query_17(`
  `const database_t& database,`

```
  int rating
```
): we calculate the overall sum of the numbers of actors playing in titles with a rating of at least `rating`; use of the `std::for_each` function and a lambda expression is expected

- ```
  std::vector<std::string> db_query_18(
    const database_t& database,
    std::string_view genre,
    std::string_view pattern,
    const std::variant<int, std::string>& rating,
    int year
  ```
  ): we find all titles that have a genre `genre`, their name matches a regular expression `pattern`, their rating is at least `rating`, they have an above-average number of actors, and all their actors were born before a year `year`; to calculate the sum of numbers of actors over individual titles, we will use the `std::accumulate` function, and calculate the resulting average as a decimal number of type `double`; for working with regular expressions, the `<regex>` library is needed; first, a regular expression instance is created via `std::regex(pattern, flags)` constructor, where a flag `std::regex_constants::icase` is used to enable the case-insensitive behavior; the actual match testing is performed using `std::regex_match(string, regex)` function; besides the classic numerical rating `0` to `100`, we will also allow ratings using stars within this query, specifically in the form of an `std::string` string containing 0 to 5 stars `*`, where the numerical value of each star is `20` points; to represent such dual ratings, we will use `std::variant` data type from the `<variant>` library; to determine the actually used variant, we will use function `std::holds_alternative<type>(variant)`, and for subsequent extraction of the corresponding value function `std::get<type>(variant)`; to find the intended titles, we will again use the `std::for_each` function; to check the condition on actor birth years, we will use the `std::all_of` function, which allows to simulate the behavior of the universal quantifier; all individual conditions, actions, and operations need to be implemented using lambda expressions this time; for each matching title, we insert a string with a JSON object `{ name: "`*title*`", year: `*year*` }` into the output container

- ```
  void db_query_19(
    const database_t& database,
    int begin,
    int end,
    std::ostream& stream = std::cout
  ```
  ): we find all titles filmed during years [`begin`, `end`); in order to retrieve them, we will use the `std::views::filter` and `std::views::transform` views, chaining them with the pipe operator `|`, where both the filtering predicate and the transformation action are to be implemented as lambda expressions; the goal of the transformation is to obtain tuples in the form `std::tuple<std::string, int, size_t>` for each title, where we store its name, year of filming, and the number of actors playing in it; we then store the obtained records in an `std::vector` container using its range constructor; subsequently, we sort all the records using the `std::ranges::sort` function, again via a lambda expression that orders the titles by their years and then by names; finally, we transform the records again using `std::views::transform`, this time into strings of the form `{ name: "`*title*`", year: `*year*`, actors: `*actors*` }`, where *title*, *year*, and *actors* are to be replaced by title name, year, and the number of actors, respectively; the resulting strings will finally be printed to the provided output stream using the `std::ranges::copy` function and a fake iterator `std::ostream_iterator` instance

- ```
  template <typename Selector, typename Comparator, typename Serializer>
  void db_query_20(
    const database_t& database,
    Selector selector,
    Comparator comparator,
    Serializer serializer,
    std::ostream& stream = std::cout
  ```
  ): we find all titles satisfying a general search condition `selector` defined as `bool operator()(const std::shared_ptr<Title>& title_ptr)`, we then sort all the matching title instances via an ordering `comparator` having interface `bool operator()(const std::shared_ptr<Title>& title_1, const std::shared_ptr<Title>& title_2)`, and finally print them serialized using a `serializer` in the
  ```

form `std::string operator()(const std::shared_ptr<Title>& title_ptr)` into a given stream `stream`

Submit again only the module with queries, i.e., files `Queries.h` and `Queries.cpp`, as well as the `Storage.h` header file with the selection of the storage type for our movie database. In particular, submit queries Q14 to Q20 only, not the older queries Q1 to Q13 or indices. Should you also submit them, nothing happens, they will just not be tested. If you have decided to use your flexible array, include its implementation, too. As for the rest of the database project, you will again not submit it.

The goal of the task is to learn to work with selected standard algorithms (`std::copy`, `std::copy_if`, `std::remove_if`, `std::sort`, `std::transform`, `std::accumulate`, `std::for_each`, `std::all_of`), as well as fake iterators (`std::back_inserter`, `std::ostream_iterator`), general lambda expressions, ranges (`std::ranges::copy`, `std::ranges::sort`), and views (`std::views::filter`, `std::views::transform`), including chaining using the `|` operator. We also get familiar with `std::variant` auxiliary data structure, and basics of regular expressions `std::regex`.

At the very end, let us briefly return to working with regular expressions. Within the Q18 query, you will receive them already formulated via a query parameter. However, if you would like to create them on your own during debugging, you will definitely benefit from the so-called raw string literals. These represent an alternative way of writing C-style strings in the form `R"`*delim*`(`*text*`)`*delim*`"`, where the delimiter *delim* can be chosen as needed (even as an empty string), and the actual resulting string is then composed solely of the *text* part content. What is essential is that within this *text* part, the standard escape sequence mechanism is not applied (e.g., `\n` remains a plain `\n`, i.e., it does not change to a newline character), which brings a significantly more user-friendly way of writing, for example, regular expressions. The reason is that they often contain many constructs expressed using backslashes `\` (e.g., `\b` to denote a word boundary, `\w` for any alphanumeric character or underscore `_`, etc.). Thanks to this, we do not have to tediously treat their occurrences, and the overall notation becomes clearer. For example, `R"(\b\wbobule\b)"` corresponds to a classic string `"\\b\\wbobule\\b"`.