NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C12**

Movie Database IV

The objective 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 (library `<algorithm>`), functors, lambda expressions, fake iterators (`<iterator>`), as well as ranges and views (`<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. Under all the circumstances, add a new header file `Storage.h` into the project, where, beside `#include` directives for all the necessary libraries, we place only one thing: definition of a type alias `database_t`, through which you explicitly choose the storage type for our database you want to use. All current code (functions for the existing queries, creation of indices, database import, etc.) will then need to be modified to work with this new alias.

Next, we add the following five new queries to our project. To truly fulfill the assignment objective, it is necessary to really use all the requested constructs and procedures.

- `std::vector<std::shared_ptr<Title>> db_query_12(`
  `const database_t& database,`
  `const Actor& actor`
  `)`: the goal is to find titles in which a given actor `actor` played; we achieve that by first copying all the titles from our database (meaning the shared smart pointers to them) into a prepared output container `std::vector`, using the `std::copy` function; 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_Q12_Actor`; finally, we 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 the `std::sort` function and a functor `Comparator_Q12_Years` simulating the behavior of the `<` operator

- `void db_query_13(`
  `const database_t& database,`
  `unsigned short year,`
  `std::ostream& stream = std::cout`
  `)`: we find all movies (titles with type `Type::MOVIE`) filmed in a particular year `year` by copying this time only the matching titles into an empty auxiliary container of 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

- `int db_query_14(`
  `const database_t& database,`
  `Type type,`
  `const std::string& genre`
  `)`: calculates the integer average rating of titles having our enumeration type `type` (with values `MOVIE` and `SERIES`) 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_Q14_Rating`; we implement it in a way that all the logic of calculating this average is encapsulated within it; for a zero count of titles, it returns an average `0`; 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_15(`
  `const database_t& database,`
  `const std::string& genre`
  `)`: calculates the overall number of titles having a genre `genre`; we will again use the `std::for_each` function, but we perform the processing of particular titles using a lambda expression this time

- `void db_query_16(`
  `const database_t& database,`
  `unsigned short begin,`
  `unsigned short end,`
  `std::ostream& stream = std::cout`
  `)`: finds all titles filmed during years [`begin`, `end`); in order to locate 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 mentioned transformation is to obtain a tuple in the form `std::tuple<std::string, unsigned short, size_t>` for each title, where we store its name, filming year, 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 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 again be printed to the provided output stream using the `std::ranges::copy` function and a fake iterator `std::ostream_iterator`

Submit only the module with queries, i.e., files `Queries.h` and `Queries.cpp`, and additionally also the new header file `Storage.h`. If you have decided to use your flexible array, include its implementation as well. As for the rest of the database project, you will again not submit it. We continue to assume that everything necessary regarding titles, movies, series, and actors is accessible through header file `Movie.h`.