NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C07**

Movie Database II

Within this homework, we return back to the topic of our database of movies. We first preserve the entire existing code and then refactor it to meet several modified and newly added requirements. In particular, we will alter the representation of our actors, we will start working with series in addition to just movies, and we will also adjust the container with which we simulate our database as such. Finally, we will also add the evaluation of three new queries, as well as adapt the existing ones to the new situation.

Let us first focus on the changes in the representation of actors. While we have represented the actors only using their atomic names until now (and, thus, technically using the `std::string` strings), we want to be capable of storing more pieces of information about them from now on, and, furthermore, in a structured way. For this purpose, we will create an `Actor` class, assuming we specifically have the following information about each actor: first name (`std::string`), last name (`std::string`), and year of birth (`int`). We will design its constructors and other methods analogously to the class for movies, in particular, we expect the following pair of constructors, as well as getter methods `name`, `surname`, and `year` for accessing the individual private data members.

- `Actor(const std::string& name, const std::string& surname, int year)`
- `Actor(std::string&& name, std::string&& surname, int year)`

In order to even be capable of working with actor instances inside the set container `std::set`, we must first define their mutual ordering. This can be achieved by implementing a global function `bool operator<(const Actor& actor1, const Actor& actor2)`, through which we introduce the behavior of the `<` operator for comparing pairs of actors. From the logical point of view, this comparison will be defined by a triple consisting of a last name, first name, and year of birth (in that order). The function returns `true` if and only if the first actor precedes the second one.

Printing an actor to a specified (or standard) output stream will be achieved by a method `void print_json(std::ostream& stream = std::cout) const`, and also via the implementation of our own operator `<<`, that is, using a global function `std::ostream& operator<<(std::ostream& stream, const Actor& actor)`. The goal is to print a given actor in the form of a JSON object, e.g., `{ name: "Ivan", surname: "Trojan", year: 1964 }`. Again, we will faithfully preserve the order of items and separators in the form of commas and spaces. We do not print any line breaks at the end.

For the purpose of importing the actors from an input stream, we add our own operator `>>`, that is, we implement a global function `std::istream& operator>>(std::istream& stream, Actor& actor)`. We expect that the individual details about a given actor are listed in the correct order and that they are separated by spaces, e.g., `Ivan Trojan 1964`. To implement such a stream extraction operator efficiently and still preserve the individual data members of the actor as private, we will use the `friend` mechanism.

The second main change in the current program consists in the fact that we will now want to work with series in addition to movies. Technically, we will make this change by first renaming the current class `Movie` to `Title`, and then deriving from it, as a common abstract ancestor, two specific classes using the inheritance, namely `Movie` for movies and `Series` for series.

In the case of movies, besides the common data items for all titles, we also want to store their optional length in minutes (`int`). For its representation and creation of values, we will use the `std::optional` type and `std::make_optional` function or `std::nullopt` constant, respectively, all from the `<optional>` library. In the case of series, on the contrary, we add their number of seasons and total number of episodes (both `int`). To distinguish between the two types of titles, we add a virtual function `Type type() const` returning a respective value from an enumeration class `enum class Type { MOVIE, SERIES }`. We will add other methods and suitable constructors as needed, assuming we will always put newly added items at the end, we will respect their order and also names of the access methods `length`, `seasons`, and `episodes`.

The existing function for printing the titles will be modified so that the first item in the generated JSON object will be an item specifying the title type, specifically `{ type: "MOVIE", … }` in the case of movies, and, vice versa, `{ type : "SERIES", … }` in the case of series. We print the actors field in the same way as the last time, but each actor will now be represented by its own JSON object, as we have already described. We will place specific title items at the end. In the case of movies, it means their length, e.g., `{ …, length:`

`112 }`. If the length is missing, we will not print the item at all. In the case of series, it means their number of seasons and episodes, e.g., `{ …, seasons: 8, episodes : 73 }`. To increase the comfort of the users, we will also offer our own stream insertion operator `<<` for titles as such.

We modify the existing function for importing movies from input CSV data analogously. We now assume that the first field for each title will be a selector determining its type, namely `MOVIE;…` for movies and `SERIES;…` for series. Again, specific fields will be given at the end, i.e., the optional length for movies, e.g., `…;112`, and the number of seasons and episodes for series, e.g., `…;8;73`. Valid ranges of the newly added numeric values are as follows: length `0` to `300`, seasons `0` to `100`, and episodes `0` to `10000`. If the first field does not contain a valid selector, we throw the following structured exception:

- Code `2` (parse errors)
  - `Invalid type selector <`*selector*`> in field <type> on line <`*line*`>`: invalid type selector value *selector*

In general, we expect that exception messages are now constructed using formatted strings `std::format` from the `<format>` library (as well as we refactor the existing exceptions from the previous assignment accordingly). When parsing the actors (i.e., directly inside the extraction operator `>>` for the actors), the detection of error situations will be based on the contextual conversion of a stream to a logical value. In particular, we assume the following structured exceptions:

- Code `2` (parse errors)
  - `Missing attribute <`*attribute*`>`: value of the first name (`name`) or surname (`surname`) attribute is missing
  - `Missing, invalid, or overflow value in attribute <`*attribute*`>`: value of the birth year attribute (`year`) is either missing, or, although present, it is not a valid number (without distinguishing the specific cause)
  - `Integer <`*value*`> out of range <`*min*`, `*max*`> in attribute <`*attribute*`>`: value of the birth year attribute is outside of the allowed range `1850` to `2100`

During the import process of movies, we then append the text messages of these exceptions (while preserving their codes) with further information describing the context of a given input movie, namely `... in actor <`*actor*`> on line <`*line*`>`, where *actor* will be replaced by the entire input string of a given actor, and *line* is again the line number. Entirely empty actors will again be skipped. Just to be sure, let us have a look at the following illustrative examples:

- `MOVIE;Pres prsty;2019;comedy;56;Petra Hrebickova 1979,Jiri Langmajer 1966;101` is correct
- `MOVIE;Pres prsty;2019;comedy;56;Petra Hrebickova 1979,Jiri;101` throws an exception with text `Missing attribute <surname> in actor <Jiri> on line <1>`
- `MOVIE;Pres prsty;2019;comedy;56;Petra Hrebickova 1979,Jiri Langmajer NaN;101` throws an exception with text `Missing, invalid, or overflow value in attribute <year> in actor <Jiri Langmajer NaN> on line <1>`

The third change will be a partially enforced modification of the representation of our simulated database as such, i.e., the container for movies. We will now need a polymorphic container that allows for working with titles of any type, i.e., movies and series. This time, however, we will no longer use traditional C-style pointers, but smart pointers, specifically in the form of the shared smart pointers. This will make the situation easier for us in terms of dynamic allocation, and, above all, deallocation of individual title instances. We therefore assume a new database of titles in the form of `std::vector<std::shared_ptr<Title>>`. Individual instances of titles will now be constructed via `std::make_shared<Movie>(…)` for movies, and analogously for series.

We will preserve both the existing database queries without changing their meaning. However, this will involve the following two technical modifications: in both of them, we will consider and so find all titles (not only the original movies), and, specifically for query `db_query_2`, considering the new structured actors, we will look for a pair of actors `Ivan Trojan 1964` and `Tereza Voriskova 1989`.

Finally, we implement the following three new queries, technically again using global functions in the same style as the last time (including printing the end of lines after each title found):

- `void db_query_3(`
  `const database_t& database,`
  `Type type, int begin, int end,`
  `std::ostream& stream = std::cout`
  `)`: we find all titles having a type `type` (movies or series according to our enumeration) that were filmed in a given interval of years [`begin`, `end`), understanding this interval as open on the right; only correct intervals of years are assumed, otherwise the behavior will be undefined; if the values `begin` and `end` are the same, a given interval is empty; for each matching title, we print a text string in the form of a JSON object `{ type: "`*selector*`", name: "`*name*`", year:` *year* `}`, where *selector* is replaced by the title type (string values `MOVIE` or `SERIES`), *name* by title name, and *year* by its year of filming

- `void db_query_4(`
  `const database_t& database,`
  `int seasons, int episodes,`
  `std::ostream& stream = std::cout`
  `)`: we find all series that have at least a given number of seasons `seasons` or at least a given number of episodes `episodes`; when retyping general titles to series (more precisely, pointers to them), it is necessary to use static retyping via `static_cast<...>(...)` or possibly also `std::static_pointer_cast<...>(...)` constructs; for each matching title, we print a text string in the form of a JSON object `{ name: "`*name*`", seasons:` *seasons*`, episodes:` *episodes* `}`, where *name* is replaced by the title name, *seasons* by the number of seasons, and *episodes* by the number of episodes

- `void db_query_5(`
  `const database_t& database,`
  `int length,`
  `std::ostream& stream = std::cout`
  `)`: we find all movies that have a length at least `length` minutes; movies that do not have their length specified will be ignored; for retyping in this case, it is necessary to use dynamic retyping instead, using `dynamic_cast<...>(...)` or possibly also `std::dynamic_pointer_cast<...>(...)` constructs, and the success of such a conversion must be properly tested; for each matching title, we again print a text string in the form of a JSON object `{ name: "`*name*`", length:` *length* `}`, where *name* is replaced by the title name and *length* by its length

As usual, submit all created source files (`*.cpp` and `*.h`) except for the main file `Main.cpp`. Within it, we again assume directives `#include` for the header files `Database.h` and `Queries.h`, through which all the expected functionality must be directly or indirectly accessible. We also continue to assume the `Storage.h` header file containing the definition of the type alias for our database.

The goal of the assignment is to verify the ability to work with selected custom operators (especially stream insertion and extraction operators), shared smart pointers used in connection with a hierarchy of classes and polymorphic containers, class `std::optional`, formatted strings `std::format`, and the mechanisms of both static and dynamic casting.

From the code quality point of view, do not forget to work with named constants. Their definitions should be placed into modules to which they logically belong, and their names should be chosen systematically (e.g., using appropriate prefixing). Simply because we already have a large number of them, and so it should be immediately clear what each constant relates to (titles, movies, series, actors, …).

During the import of titles, it is essential to process individual actors using their stream extraction operator `>>`. This was actually the very reason why we have designed it in the first place. Inside this operator, the first name, surname, and birth year attributes should again be processed using their respective extraction operators, and the obtained values should be stored directly into the provided output actor instance. Finally, let us note that movies themselves could be handled analogously via their own extraction operator as well. However, due to the CSV format (and so separators other than whitespace characters), this would be more complicated and without meaningful benefits for us. Hence, such a change is not worth implementing.

When importing and printing specific variants of titles, i.e., movies and series, avoid duplicating the same or similar code. In other words, shared attributes (those belonging to every title) should be handled once, not repeatedly. In the case of printing the titles as JSON objects, make use of the benefits offered

by the mechanism of virtual methods. Even though general titles are abstract, their hypothetical printing should still result in a well-formed and meaningful JSON output.

Since we are now familiar with string views `std::string_view`, consider whether it would be reasonable to use them in suitable places (aside from the binding interface, i.e., only in your internal methods, whether newly introduced or by refactoring the existing code). At the very end, let us once again recall our requirement regarding the database query `db_query_2`, where verification of the presence of actors must still be accomplished in logarithmic time.