NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C07**

Movie Database II

Within this homework, we will return back to the topic of our database of movies. We first preserve the entire existing code and then refactor it to meet a couple of modified and newly added requirements. In particular, we will alter the representation of our actors, we will start working with series in addition to movies, and we will also adjust the container with which we simulate our database as such. Finally, we will also add the evaluation of two 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 (`unsigned short`). 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 methods `name`, `surname`, and `year` for accessing the individual data items.

- `Actor(const std::string& name, const std::string& surname, unsigned short year)`
- `Actor(std::string&& name, std::string&& surname, unsigned short 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 will 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.

Printing of 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 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 length in minutes (`unsigned short`), and in the case of series, on the contrary, their number of seasons and total number of episodes (both `unsigned short`). 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 }`, in the case of series, their number of seasons and episodes, e.g., `{ …, seasons: 8, episodes : 73 }`. To increase the comfort of users, we will also offer our own stream insertion operator `<<` for titles as such.

We will modify the existing function for importing movies from input CSV files 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 length for movies, e.g., `…;112`, and the number of seasons and episodes for series, e.g., `…;8;73`. If the first field does not contain a valid selector, we return our structured exception with code `2` and text `Invalid type selector <`*selector*`> in field <type> on line <`*line*`>` with an appropriate line number. Valid ranges of newly added numeric values are as follows: length `0` to `300`, seasons `0` to `100`, and episodes `0` to `10000`.

When parsing actors, we skip entirely empty actors. The detection of error situations will be based on the behavior of the stream conversion to a logical value, and, therefore, for attributes `name` and `surname`, we will return an exception `Missing attribute <`*attribute*`> in actor <`*actor*`> on line <`*line*`>`, and for birth year attributes `year` only a single merged exception `Missing, invalid, or overflow value in attribute <year> in actor <`*actor*`> on line <`*line*`>` or subsequently an exception for a value outside of the allowed range `1850` to `2100` in the form `Integer out of range <`*min*`, `*max*`> in attribute <year> in actor <`*actor*`> on line <`*line*`>`. In all these cases, it will be our structured exceptions with code `2`, where *actor* will be replaced by the entire input string of a given actor. Let us also mention that we do not have to create the text content of the mentioned exceptions all at once, and so we can generate its part directly in the `>>` operator function and then add the rest later on.

To be sure, let us have a look at the following three 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>`, and
- `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>`.

Finally, the third change will be a partially enforced modification of the representation of our simulated database as such, i.e., the movie container. 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 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 all titles (not only the original movies), and, specifically, for query `db_query_2`, following the new structured actors, we will look for a pair of actors `Ivan Trojan 1964` and `Tereza Voriskova 1989`.

Finally, we implement two 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 std::vector<std::shared_ptr<Title>>& db,
    unsigned short seasons, unsigned short episodes,
    std::ostream& stream = std::cout
  ```
  ): we find all series that have at least the specified number of seasons `seasons` or at least the specified number of episodes `episodes`; we will print the entire JSON objects of the found titles

- ```
  void db_query_4(
    const std::vector<std::shared_ptr<Title>>& db,
    const std::type_info& type, unsigned short begin, unsigned short end,
    std::ostream& stream = std::cout
  ```
  ): we find all titles of the specified type (movie or series) that were filmed in the specified interval of

years [`begin`, `end`), understanding this interval as open on the right; to determine a given title type `type`, we will use the information type classes `std::type_info` from the library `<typeinfo>` and a function `typeid(…)`, which can be called over classes as well as their instances; for each suitable title, we only print its name; let us note that 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

As usual, submit all the created source files (`*.cpp` and `*.h`) except for the main file `Main.cpp`. Within it, we will again assume directives `#include` for the header files `Database.h` and `Queries.h`. The goal of the task is to verify the ability to implement selected custom operators as well as to work with shared smart pointers used in conjunction with a hierarchy of classes and polymorphic containers.