

Movie Database I

The goal of this task is to implement the first part of an application that will allow us to simulate work with a database of movies. By this we mean the ability to represent and store individual movies (create their instances and store them in a standard vector) and then query them (search for movies that match specific search criteria and then print them to the standard output).

We need to store the following data items for each movie: movie name (of type `std::string`), year of filming (`int`), genre (`std::string`), rating (`int`), and a set of names of actors (`std::string`) who played in a given movie. The set of these actors will be realized using another type of a standard container, namely `std::set`, which can be found in the `<set>` library. All data members are mandatory in general, but the number of actors can be completely arbitrary, even zero.

In order to represent the described movies, we create an ordinary class named `Movie`, each specific movie will then be represented by one instance of this class. All the above-mentioned pieces of data will be implemented as private data members, access to them from the outside will only be possible via public parameterless getter methods `name`, `year`, `genre`, `rating`, and `actors`, all returning unmodifiable references to the relevant items (in the case of title, genre, and actors) or copies of the respective values (for year and rating). In addition, these methods must be declared as `const` (so that we can call them even on non-modifiable movie instances), and we will program them as inline methods for the sake of efficiency.

The `Movie` class will offer the following two constructors. The first of them will save copies of the passed items, the second one will appropriate the items passed by rvalue references. It is expected that the mechanism of member initializer lists will be used to store the individual items in both the cases.

- `Movie(const std::string& name, int year, const std::string& genre, int rating, const std::set<std::string>& actors)`
- `Movie(std::string&& name, int year, std::string&& genre, int rating, std::set<std::string>&& actors)`

The last method of the `Movie` class will be `void print_json(std::ostream& stream = std::cout)`. Through it, we will be able to print a given movie to the provided output stream. In particular, we will print it in the form of a JSON object, its structure must correspond to the following template:

```
{ name: "Bobule", year: 2008, genre: "comedy", rating: 65, actors: [ "Krystof Hadek", "Te
reza Voriskova" ] }
```

In addition to spaces and the overall structure, we must also observe the order and names of the individual items and enclose text values in double quotes. We will enumerate the names of the actors in exactly the same order as they are returned to us by the relevant set container iterator. If a given movie does not have even a single actor, we will then not include the `actors` property at all. Otherwise, of course, we do not print the separating comma after the very last of them inside the array. There are no line breaks involved in the output, not even at the end after the closing curly brace of the entire JSON object. Furthermore, we assume that values of the movie titles and genres, and actor names do not contain double quotes. Therefore, we do not need to treat their possible occurrences in order to ensure the correct structure of the output.

The database itself will be represented using the standard vector container `std::vector`, into which we will gradually insert individual movie instances. However, in order to prepare for changes that await us in the follow-up assignments, we will not work with this type directly. Instead, we will consider a type alias in the form `database_t = std::vector<Movie>`. This will allow us to easily change the representation of movies as such right in the next assignment, and even replace the vector container with another one at the end of the semester. The definition of this type alias (as the only thing, of course apart from the necessary `#include` directives) will be placed in a header file named `Storage.h`.

For inserting individual movies into our current database, we can use, for example, the `push_back` or `emplace_back` methods. However, we will also offer the users the possibility of batch movies import from input files or general input streams. We propose a `Database` class for this purpose, assuming that its definition will be placed in a `Database.h` header file. This class is not intended to store the content of our database, it will only encapsulate the import functionality through the following two public static methods:

- `static void import(const std::string& filename, database_t& database):`
opens a given input file, imports the movies, and inserts their instances into the prepared database (while preserving its possible current content)
- `static void import(std::istream& stream, database_t& database):`
performs the import of movies from a given input stream and inserts their instances into the prepared database

In both cases, we assume the input data to be in the CSV format (without a header, i.e., without the first line containing column names). This means that there will be one movie on each individual line, entirely empty lines will be skipped. For a given movie, we gradually expect its title, year, genre, rating, and names of actors. The order of these fields is fixed and implicit, and so we will hard-wire them in our code directly. Individual fields will be separated (not terminated) by a semicolon, names of individual actors by a comma. These separators cannot appear anywhere inside the values, excessive line content will be ignored.

Movie name and genre fields must be non-empty strings. As for the allowed values of numeric fields, year of filming is expected to be from the closed interval 1900 to 2100, movie rating from 0 to 100. Since we will add several more fields in the follow-up assignment, it is necessary to decompose the code appropriately and strictly avoid any code repetition. This means at least two functions to process string and numeric fields (including low-level retrieval of their value, handling error situations, and then checking the listed integrity constraints). As for the actors, it will be more appropriate to process them specifically (because we will change their internal structure in the next task), empty actors will be skipped. Let us also recall that the number of actors can be zero, as can be seen in the second movie within the following input example:

```
Dira u Hanusovic;2014;comedy;49;Tatiana Vilhelmova,Ivan Trojan,Klara Meliskova
Vlastnici;2019;comedy;74;
```

Use of the global function `std::getline` is expected for the input parsing as such, this time also in the variant with an optional third parameter, via which a separator other than the default line endings can be specified. For this purpose, we will need to be capable of creating streams over strings, using `std::istringstream` from the `<sstream>` library. We will insert movie instances into the database container as efficiently as possible, i.e., using the already mentioned `emplace_back` method (which expects parameters of some constructor of our movie, not an already created instance of such a movie) together with the `std::move` construct.

We will handle all error situations using structured exceptions, in the form of a structure `struct Exception { int code; std::string text; }`, where the first item will contain the numeric code of the error type and the second one a text string explaining the cause of the error in more detail. Specifically, we expect the following behavior:

- Code 1 (input errors)
 - Unable to open input file *<filename>*:
it is not possible to open an input file with name *filename*
- Code 2 (parse errors)
 - Missing field *<name>* on line *<line>*:
the respective field is missing, i.e., there is no other field in the input string (e.g., field `rating` in input `Pres prsty;2019;comedy`)
 - Empty string in field *<name>* on line *<line>*:
empty string for a text field that does not permit empty values (e.g., field `genre` in input `Pres prsty;2019;;56;Petra Hrebickova,Jiri Langmajer`)
 - Invalid integer *<value>* in field *<name>* on line *<line>*:
invalid value *value* for number fields as a response to the `std::invalid_argument` exception from the `std::stoi` function (e.g., field `year` in input `Pres prsty;NaN;comedy;56;`)
 - Overflow integer *<value>* in field *<name>* on line *<line>*:
analogously, overflowed number value *value* as a response to the `std::out_of_range` exception
 - Malformed integer *<value>* in field *<name>* on line *<line>*:
analogously, not well formed numeric value *value* as a response to a failed position test (e.g., field `year` in input `Pres prsty;2019ad;comedy;56;`)

- Integer *<value>* out of range *<min, max>* in field *<name>* on line *<line>*:
valid numeric value *value* out of the closed range of permitted values [*min*, *max*]

Parameters in angle brackets will be replaced by particular values, the brackets themselves will be preserved: *filename* is a name of the required file, *value* is the original parsed value, *name* is a name of the problematic field (i.e., **name**, **year**, **genre**, **rating**, or **actors**), *line* is a line number in the input file (counting from 1, including empty lines), and *min* and *max* is the interval of the allowed values for numeric fields.

If we encounter any problems when parsing a particular movie record, we will not instantiate that movie, we will throw the appropriate exception, and so we will not continue processing anything that could remain in the input. All movies that we have already managed to insert into the database will remain in it untouched. To construct text messages of exceptions, it suffices to use the concatenation operator `+` on strings (at least one operand must be of type `std::string`, use function `std::to_string` from the `<string>` library to convert numbers).

Finally, we will implement two global functions with which we will simulate evaluation of particular queries over our database. In both cases, we will simply iterate through all the movies in the passed container and print the ones we were looking for in the intended form into the specified output stream. Each such movie found will also be terminated by the end of line. Both the functions will be declared in a header file `Queries.h`.

- `void db_query_1(const database_t& database, std::ostream& stream = std::cout):`
we find all movies (i.e., no filtering is performed); we print each of them as a complete JSON object (using the function we prepared)
- `void db_query_2(const database_t& database, std::ostream& stream = std::cout):`
we find all comedies (movies with genre *comedy*) filmed before year *2010*, in which *Ivan Trojan* or *Tereza Voriskova* played; for each of them, we print only the corresponding movie titles; we will hard-wire all the mentioned query parameters directly in the code, the possibility of their eventual change is not expected due to the experimental nature of this function (i.e., we will specifically not create any named constants for these values, as it would not make sense); evaluation of the condition on actors must be performed in logarithmic time; therefore, it is not acceptable to iterate through all the actors sequentially and compare them individually using manual equality tests

Submit all created source files (`.cpp` and `.h`) except for the main file with the `main` function. This time again, the prepared test already contains it and it will control the course of the entire test. All code should be split into appropriately separated modules. As already stated, the `Database` class must be defined in the `Database.h` header file, functions `db_query_*` must be declared in the `Queries.h` file. All the expected functionality must then be directly or indirectly accessible through these two header files only.

The particular goal of the task is to verify the ability to work with the following constructs: design and use of ordinary data classes, design of their parameterized constructors in combination with member initializer lists, design of inline functions, use of the `std::move` construct and work with rvalue references, getting acquainted with the `std::set` container and advanced ways of inserting elements into the `std::vector` container using the general `emplace` mechanism, and, finally, working with the `std::getline` function with a separator different to the default one, combined with `std::istringstream` streams. It is necessary to use all these constructs within your solution.

It is assumed that all the general requirements we already have for the assignments will be followed. Specifically, do not forget named constants for names of movie attributes (they are the same in JSON objects as well as in exception texts), for exception codes, and for the semicolon and comma delimiters we use in the input CSV files (so that they can easily be changed if need be). On the contrary, do not create constants for the metasympols used in JSON objects (they are fixed and cannot be changed). Also, do not forget to use the convention of postfixing names of private data members in our classes with the `_` symbol.

Each local variable is supposed to be declared in the most specific block, i.e., only in such a block (which is always defined by a pair of curly braces `{}`) where we really need to use it. Any other auxiliary methods for parsing movies should again be added as private static member functions of the `Database` class, not as standalone global functions. Inline functions must be programmed in a way that not only their declarations but also their full definitions are known at the places of use. In our case, this means that we have to implement their bodies directly in header files.

When parsing individual fields of a particular movie from the input, it would not be appropriate to use a loop (either `while` or `for`) since we need to process each field differently. In other words, loops are generally only useful in situations where each iteration proceeds, let us say, similarly, or at least has some sufficiently significant common ground. In our case, however, we would have to re-branch the body of the loop for each individual case (perhaps using the `switch` construct or otherwise) and solve them separately. Regardless of the obvious inefficiency, such a solution would primarily be a bad design. We also avoid an approach where we would first extract all the fields and store them as strings in some temporary container before their further processing.

In general, we assume that input values are checked during the import, hence movie constructors already expect valid values and do not perform any checks. Exceptions responding to error situations should be thrown directly at the point of detection and not be wrapped in separate functions. That would only confuse the reader of our code (they would conceal the intention of throwing these exceptions) and we would probably call each of them only once anyway. Finally, let us realize that it is also not advisable to propose systematic functions for printing the JSON objects, arrays, and values, because they can generally nest into each other recursively without any limitation.