Systems Programming – Assignment #1
Streaming Service System
2020/Fall

1 Before You Start

- It is mandatory to submit all the assignments in pairs. It is recommended to find a partner as soon as possible and create a submission group in the submission system. Once the submission deadline has passed, it will not be possible to create submission groups even if you have an approved extension.
- Read the assignment together with your partner and make sure you understand the tasks. Please do not ask questions before you have read the whole assignment.
- Skeleton classes will be provided on the assignment page, you must use these classes as a basis for your project and implement (at least) all the functions that are declared in them.

KEEP IN MIND

While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. It is your own responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade 0 to your assignment.

Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine before submitting it.

We will reject, upfront, any appeal regarding this matter!!

We do not care if it runs on any other Unix/Windows machine.

Please remember, it is unpleasant for us, at least as it is for you, to fail your assignments, just do it the right way.

2 Assignment Goals

The objective of this assignment is to design an object-oriented system and gain implementation experience in C++ while using classes, standard data structures and unique C++ properties such as the “Rule of 5”. You will learn how to handle memory in C++ and avoid memory leaks. The resulting program must be as efficient as possible.

3 Assignment Definition

In this assignment you will write a C++ program that simulates a new streaming service - SPLFLIX. SPLFLIX offers users two types of streaming content – movies and tv-episodes, which belong to a given tv-series. Like most streaming services we allow a creation of multiple users, while each user receives user-specific recommendations, according to his watch history.

Figure 1: A person using SPLFLIX

However, on SPLFLIX, unlike current streaming services, we give our users the ability to choose which recommendation algorithm to use, out of a number of algorithms we offer. Each user stores its own watch history, and can use a different recommendation algorithm. After the user chooses to watch a content, a recommendation can be given to the user for the next content.

At each point during the session, there is only one active user (while other users are considered as non-active), which means- only one user can watch SPLFLIX at any given point.

The program will receive a config file (json) as an input, which includes all the information about the available streaming content (see section 3.5).

3.1 The Program flow

The program receives the path of the config file as the first command line argument. Once the program starts (which should trigger the *start()* function), it opens the SPLFLIX by printing “SPLFLIX is now on!” to the screen, and initializing a default watching user. The default user

has the name "default", and its preferred recommendation algorithm is the “Length recommender” (as described in section 3.3).

Afterwards, the program enters a loop, in which it wats for the user to enter an action to execute. After each executed action, it should wait for the next action.

The loop ends when the user enters the action "exit". (See actions in part 3.4)

An important note – we will test your program on various settings, make sure that in your implementation, the session can be started again after the user enters the exit command. That is – the exit() command should only exit the main loop, and shouldn't change any of your other data structures. If one activates the start() method again after the user entered the exit command – the expected behavior is that the session would resume running.

3.2 Classes

Session – This class holds all the information that is relevant to a session: a list of users, currently active user, available watching content, and history of all actions (actions log).

Watchable – This abstract class represents a watchable content. Each watchable content has an id, a name, length, and a list of tags that describes it. It has the pure virtual method *getNextWatchable(const Session&)*, which returns a pointer to the watchable content to be recommended to the user, or a null pointer, in the case in which there is no recommendation.

Movie – Inherits from Watchable. This class represents a standalone movie.

Episode – Inherits from Watchable. This class represents an episode of a tv series.

User – This is an abstract class for the different user classes. There are several types of users, which differs by their preferred recommendation algorithms. Each class that is derived from this class implements a different recommendation algorithm (as described in section 3.3). This class stores the user’s name, and his watch history. It has a pure virtual function *getRecommendation()*, which returns the recommended movie/episode, according to its recommendation algorithm.

BaseAction – This is an abstract class for the different action classes. It has the pure virtual method *act(Session& sess)* which receives a reference to the current session as a parameter

and performs an action on it, a pure virtual method *toString()* which returns a string representation of the action, and a flag status, which stores the current status of the action: “PENDING” for actions that weren't performed yet, “COMPLETED” for successfully completed actions, and “ERROR” for actions which couldn't be completed.

After each action is performed, its status should be updated: if the action was completed successfully, the protected method *complete()* should be called in order to change the status to “completed”. If the action resulted in an error, then the protected method *error(std::string& errorMsg)* should be called in order to change the status to “error” and update the error message.

When an action results in an error, the program should print to the screen:

“Error - <error_message>”

More details about the actions will be provided in section 3.4.

Figure 2: With SPLFLIX - You no longer need to visit the cinema!

3.3 Recommendation Algorithms

All algorithms should apply the following common behavior:

- After watching a movie, the next recommendation should be dictated by the user's recommendation algorithm (which could be either a movie or an episode).
- After watching an episode, the next recommendation should be the next episode in the series. If there is no such episode, the recommendation should be dictated by the user's recommendation algorithm (which could be either a movie or an episode).

Algorithm types:

Length Recommender – This algorithm is based on the assumption that users prefer to watch content in a similar length. It will recommend the content whose length is closest to the average length of the content in the user's watch history, and which **isn't** in the user's watch history. (3-letter code – len)

Rerun Recommender – This algorithm is intended for users who don't like new stuff. It will recommend content based on this user's watching history. Starting from the first watchable content, in a cyclic order. That is – the first watchable to be recommended is the first watchable watched by the user. Afterwards, if the last watchable that was recommended by the algorithm was at index i in the history, and the history is of length n the next content to be recommended is at index $(i+1) mod\ n$.(3-letter code – rer)

Similar Genre – This algorithm will recommend content based on the most popular tag in the user's watch history. If a set of tags has the same popularity, the algorithm will use lexicographic order to pick between them. It will recommend a content which is tagged by the most popular tag in the user's history, which wasn't already watched by this user. If no such content exists, it will try with the second most popular tag, and so on. (3-letter code – gen)

Notes:

- All algorithm rules are applied on all watchable content (movies and episodes together).

- In the case in which there is more than one content which fits the recommendation criteria – for example, two movies whose length is equal to the average content length in the user's watching history, the content with the smaller index in the content vector would be picked.
- If no content fits the recommendation criteria – a null pointer should be returned.

3.4 Actions

Below is the list of all actions that can be requested by the user. Each action should be implemented as a class derived from the class BaseAction.

- **Create User –** Creates a new user in the current session.
  Syntax: createuser <user_name> <recommendation_algorithm>
  Where the <recommendation_algorithm> is the 3-letter code for that algorithm (as described in section 3.3).
  If the 3-letter code is invalid, or there is already a user with that name, this action should result in an error.
  Example:
  " createuser Yossi len" – Will create a new user instance, with the name Yossi, and its recommendation algorithm will be the Length Recommender.

- **Change Active User–** Changes the current active user.
  Syntax: changeuser <user_name>
  If the user doesn't exist, this action should result in an error.
  Example:
  " changeuser Yossi" – Changes the current active user to be Yossi's.

- **Delete User** – Removes a given user.
  Syntax: deleteuser <user_name>
  If the user doesn't exist, this action should result in an error.
  Example:
  "deleteuser Yossi" – Deletes Yossi's user.

- **Duplicate User** – Creates a copy of the given user, with its watch history and recommendation algorithm, but with a new name.
  Syntax: dupuser <original_user_name> <new_user_name>
  If the original user doesn't exist, or the new user name is already taken, this action should result in an error.
  Example:
  "dupuser Yossi David" – Creates a copy of Yossi's user, with the name David.

- **Print Content List** – Prints all the watchable content available.
  Syntax: content
  Each watchable content should be printed in its own line in the following manner:
  <content_id> <content_name> <content_length> minutes [<tag_1>, <tag_2>, …, <tag_n>]
  Where <content_name> is given by the content's *tostring()* method, and <content_id> is the location of the given content in the content vector.
  Example:
  "content" will print:

  1. Game of Thrones S01E01 62 minutes [Fantasy, Drama]
  2. Game of Thrones S01E02 56 minutes [Fantasy, Drama]
  3. Inglorious Basterds 153 minutes [War, Western]
  4. The Shawshank Redemption 142 minutes [Drama]

- **Print Watch History** – Prints the watch history of the current active user.
  Syntax: watchhist
  The first line to be printed should be:
  Watch history for <user_name>
  Where <user_name> is the name of the current active user.
  After the first line, every watchable content in the history of this user should be printed in its own line in the following manner:
  <id> <content_name>

Where <content_name> is given by the content's *tostring()* method, and <id> is the index in the history vector, where 1 is the earliest watchable item.

Example:

"watchhist" will print:

```
Watch history for Yossi
1. Game of Thrones S01E02
2. Inglorious Basterds
```

- **Watch** – Watches a content.
  Syntax: watch <content_id>
  Where <content_id> is the id of the content in the content vector.
  After this command, the line:
  "Watching <content_name>"
  should be printed, where <content_name> is given by the content's *toString()* method.
  After watching, the content should be added to the watch history of the current user, and a recommendation for a new content should be given to the user, by printing the following line to the screen:
  “We recommend watching <content_name>, continue watching? [y/n]”
  If the user chooses yes, a new watch command should be executed (implicitly) for the recommended content.
  Example:
  "watch 1" Will print:

```
Watching Game of Thrones S01E01
We recommend watching Game of Thrones S01E02, continue watching? [y/n]
```

- **Print Actions Log** – Prints all the actions that were performed by the user (Excluding current log action), from the last one to the first, in the following syntax:
  <order_n_tostring> <order_n_status>
  …
  <order_1_tostring> <order_1_status>

where the status of each order is "completed" if the order was completed successfully, or "error: <error_message>" otherwise.

Syntax: log

Example:

```
DuplicateUser ERROR: the new user name is already taken
Watch COMPLETED
CreateUser COMPLETED
```

- **Exit** – Exits the main loop.

  Syntax: exit

3.5 Configuration file format

The input file is of JSON format and contains the content available in the current SPLFLIX session. The input file contains a list of movies, to each a name, a length, and a list of tags. And a list of tv series – To each a name, the length of an episode, a list of seasons, which stores the number of episodes in each season, and a list of tags.

During initialization, the program should read the input file, create the watchable objects accordingly, and insert them to the content vector in class session. The order in which the objects should be inserted to the vector is according to their order of appearance in the JSON file, where movies should appear before tv-episodes.

In order to read JSON format with C++, we supplied you with Niels Lohman's JSON for Modern C++. You can learn how to use this package, and see examples here:

https://github.com/nlohmann/json

3.6 General instructions

- Class with resources must implement the rule of five.
- You may not change the interface of the objects. i.e. you may not modify the function declaration nor add any public/protected methods/data members. See section 4 for more details.
- You may not add any global variables.

4 Provided files

The following files will be provided for you on the assignment homepage:

Session.h

Watchable.h

Action.h

User.h

Main.cpp

You are required to implement the supplied functions and to add the Rule-of-five functions as needed. You are **NOT ALLOWED** to modify the signature (the declaration) of any of the supplied functions. We will use these functions to test your code, therefore any attempt to change their declaration might result in a compilation error and a major deduction of your grade.

You also **must not** add any global variables to the program.

5 Submission

- Your submission should be in a single zip file called ”student1ID-student2ID.zip”. The files in the zip should be set in the following structure:
  - src/
  - include/
  - bin/
  - makefile

  src/ directory includes all .cpp files that are used in the assignment.
  Include/ directory includes the header (.h or *.hpp) files that are used in the assignment.
  bin/ directory should be empty, no need to submit binary files. It will be used to place the compiled file when checking your work.

- The makefile should compile the cpp files into the bin/ folder and create an executable named ”splflix” and place it also in the bin/ folder.
- Your submission will be build (compile+link) by running the following commands: "make".
- Your submission will be tested by running your program with different scenarios.
- Your submission must compile without warnings or errors on the department computers.
- We will test your program using VALGRIND in order to ensure no memory leaks have occurred. We will use the following valgrind command:
  valgrind --leak-check=full --show-reachable=yes [program-name] [program parameters].
  The expected valgrind output is:
  definitely lost: 0 bytes in 0 blocks
  indirectly lost: 0 bytes in 0 blocks
  possibly lost: 0 bytes in 0 blocks
  suppressed: 0 bytes in 0 blocks
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

  We will ignore the following error only:
  still reachable: 72,704 bytes in 1 blocks (known issue with std). **We will not ignore** "still reachable" with different values than **72,704** bytes in **1** blocks

- Compiler commands must include the following flags:
  -g -Wall -Weffc++ -std=c++11.

6 Recommendations

1. Be sure to implement the rule-of-five as needed. We will check your code for correctness and performance.
2. After you submit your file to the submission system, re-download the file that you have just submitted, extract the files and check that it compiles on the university labs. Failure

to properly compile or run on the departments' computers will result in a zero grade for the assignment.

בהצלחה!