# CMP-5015Y Coursework 3 - Offline Movie Database in C++

100250071 (`fxg18asu`)

Monday 11th May, 2020 13:24

PDF prepared using LaTeX template v1.00 .

☑ I agree that by submitting a PDF generated from this template I am confirming that I have checked the PDF and that it correctly represents my submission.

## Contents

# Movie.h

```
/**
 * File - Movie.h
 * Author - Luke Marden
 * Date - 09/04/2020
 * Description - An object to handle a films data
 */

#ifndef INC_5015Y_CW2_MOVIE_H
#define INC_5015Y_CW2_MOVIE_H

#include <functional>
#include <iostream>
#include <string>

class Movie {
private:
    /**
     * All the attributes of the movies
     */
    std::string name, certificate, genre;
    int year, duration, rating;
public:
    /**
     * The constructor
     * @param name - the movies name
     * @param year - the year the movie came out
     * @param certificate - the age certificate of the movie
     * @param genre - the genres of the movie
     * @param duration - the duration of the movie in minutes
     * @param rating - the rating of the movie
     */
    Movie(std::string name, int year, std::string certificate, std::string genre,
        int duration,
            int rating);
    /**
     * A method to get the movies name
     */
    inline std::string getName(){
        return this->name;
    }
    /**
     * A method to get the movies year
     * @return the year
     */
    inline int getYear(){
        return this->year;
    }
    /**
     * A method to get the movies certificate
     * @return the movies certificate
     */
    inline std::string getCertificate(){
        return this->certificate;
    }
    /**
     * A method to get the movies genres
     * @return the movies genres
     */
    inline std::string getGenre(){
        return this->genre;
    }
```

```
61      /**
         * A method to get the movies duration
63       * @return the movies duration
         */
65      inline int getDuration(){
            return this->duration;
67      }
        /**
69       * A method to get the movies rating
         * @return the movies rating
71       */
        inline int getRating(){
73          return this->rating;
        }
75      /**
         * A method to set a movies name
77       * @param name - the name to set it as
         */
79      inline void setName(std::string name){
            this->name = name;
81      }
        /**
83       * A method to set the movies year
         * @param year - the year to set it as
85       */
        inline void setYear(int year){
87          this->year = year;
        }
89      /**
         * A method to set the movies certificate
91       * @param certificate - the certificate to set it as
         */
93      inline void setCertificate(std::string certificate) {
            this->certificate = certificate;
95      }
        /**
97       * A method to set the movies genres
         * @param genre - the genres to set it as
99       */
        inline void setGenre(std::string genre){
101         this->genre = genre;
        }
103     /**
         * A method to set the movies duration
105      * @param duration - the duration to set it as
         */
107     inline void setDuration(int duration){
            this->duration = duration;
109     }
        /**
111      * A method to set the movies rating
         * @param rating - the rating to set it as
113      */
        inline void setRating(int rating){
115         this->rating = rating;
        }
117     /**
         * A method to output the details of a movie
119      * @param output - the output stream
         * @param movie - the movie to output
121      * @return a series of strings that represent the movies data
         */
123     friend std::ostream& operator<<(std::ostream &output, const Movie &movie);
```

```
125
};
127 /**
 * An overridden operator that checks if a movie is equal to another
129  * @param m1 - the first movie
 * @param m2 - the second movie
131  * @return true if they are the same, false if they are not
 */
133 bool operator==(Movie& m1, Movie& m2);
/**
135  * An overridden operator that checks if a movie is not equal to another
 * @param m1 - the first movie
137  * @param m2 - the second movie
 * @return true if they are not the same, false if they are the same
139  */
bool operator!=(Movie& m1, Movie& m2);
141
#endif //INC_5015Y_CW2_MOVIE_H
```

# Movie.cpp

```
/**
 * File - Movie.cpp
 * Author - Luke Marden
 * Date - 09/04/2020
 * Description - An object to handle a films data
 */
#include <functional>
#include <iostream>
#include <iomanip>
#include <sstream>
#include "Movie.h"
/**
 * The constructor
 * @param name - the movies name
 * @param year - the year the movie came out
 * @param certificate - the age certificate of the movie
 * @param genre - the genres of the movie
 * @param duration - the duration of the movie in minutes
 * @param rating - the rating of the movie
 */
Movie::Movie(std::string name, int year, std::string certificate, std::string
    genre, int duration, int rating) {
     this->name = name;
     this->year = year;
     this->certificate = certificate;
     this->genre = genre;
     this->duration = duration;
     this->rating = rating;
}

/**
 * An overridden operator that checks if a movie is equal to another
 * @param m1 - the first movie
 * @param m2 - the second movie
 * @return true if they are the same, false if they are not
 */
bool operator==(Movie& m1, Movie& m2){
    return (m1.getName() == m2.getName() && m1.getYear() == m2.getYear() &&
        m1.getCertificate() == m2.getCertificate() && m1.getGenre() == m2.
            getGenre()
        && m1.getDuration() == m2.getDuration() && m1.getRating() == m2.getRating
            ());
}
/**
 * An overridden operator that checks if a movie is not equal to another
 * @param m1 - the first movie
 * @param m2 - the second movie
 * @return true if they are not the same, false if they are the same
 */
bool operator!=(Movie& m1, Movie& m2){
    return !(m1 == m2);
}
/**
     * A method to output the details of a movie
     * @param output - the output stream
     * @param movie - the movie to output
     * @return a series of strings that represent the movies data
     */
std::ostream& operator<<(std::ostream &output, const Movie &movie){
    output << "\"" << movie.name << "\",";
    output << movie.year << ",\"";
```

```
    output << movie.certificate << "\",";
    output << "\"" << movie.genre << "\",";
    output << movie.duration << ",";
    output << movie.rating;
    return output;
}

//std::istream &operator>>(std::istream& input, Movie& movie){
//    std::string name, classification, genre;
//    int year, length, rating;
//    std::cout << "What is the name of the film?\n";
//    input >> name;
//    std::cout << "What year was the movie made?\n";
//    input >> year;
//    std::cout << "What's the age rating of the movie?\n";
//    input >> classification;
//    std::cout << "What's the genre of the movie?\n";
//    input >> genre;
//    std::cout << "What's the duration of the movie?\n";
//    input >> length;
//    std::cout << "What's the rating of the movie?\n";
//    input >> rating;
//    movie = Movie(name, year, classification, genre, length, rating);
//    return input;
//}
```

# MovieDatabase.h

```
/**
 * File - MovieDatabase.h
 * Author - Luke Marden
 * Date - 21/04/2020
 * Description - A class to handle a database of movie objects
 */
#include <vector>
#include <fstream>
#include <sstream>
#include <iosfwd>
#include <string>
#include <iostream>
#include <regex>
#include "Movie.h"
#ifndef INC_5015Y_CW2_MOVIEDATABASE_H
#define INC_5015Y_CW2_MOVIEDATABASE_H
enum SortDirection {ASC, DESC};
enum Attribute {NAME, NAMELENGTH, YEAR, CERTIFICATE, GENRE, DURATION, RATING};

class MovieDatabase {
public:

    /**
     * Constructor
     */
    MovieDatabase();
    /**
     * Destructor
     */
    ~MovieDatabase();
    /**
     * Method to add movies to the database
     * @param movie - the movie to add
     */
    void addMovie(Movie *movie){
        this->db.push_back(movie);
    }
    /**
     * Makes a sub database based on an attribute and a value
     * @param attribute - the attribute to build the sub database on
     * @param value - the value to match
     * @return The sub database
     */
    MovieDatabase* subDB(Attribute attribute, std::string value);
    /**
     * Sorts the database based on a movies attribute and a direction
     * @param attribute - Movie name etc
     * @param direction - Ascending/Descending
     */
    void sortDB(Attribute attribute, SortDirection direction);
    /**
     * Method to get the db of the object
     * @return the database vector
     */
    inline std::vector<Movie*> getDB(){
        return this->db;
    }
    /**
     * Gets the size of the database
     * @return the size of the database
     */
```

```
    inline int getSize(){
63      return db.size();
    }
65  /**
     * Gets the movie at the index specified
67   * @param index - the index to find the movie at
     * @return the movie at that index
69   */
    inline Movie* get(int index) {
71      return db[index];
    }
73  /**
     * Overidden ostream method to output every movie in the database
75   * @param output - the output stream
     * @param db - the database
77   * @return a series of strings to the console that display all the movies in
         the
     *          database
79   */
    friend std::ostream &operator<<(std::ostream& output, const MovieDatabase &db
       );
81

83 private:
    /**
85   * A vector to store the movie objects
     */
87  std::vector<Movie*> db;
    /**
89   * Comparator to see which movie is bigger based on a films attributes
     * @param m1 - movie 1
91   * @param m2 - the movie to compare movie 1 to
     * @param attribute - the movies attribute to compare
93   * @return true if m2 is bigger than m1
     */
95  bool sortAscending(Movie* m1, Movie* m2, Attribute attribute);
    /**
97   * Comparator to see which movie is bigger based on a films attributes
     * @param m1 - movie 1
99   * @param m2 - the movie to compare movie 1 to
     * @param attribute - the movies attribute to compare
101  * @return true if m1 is bigger than m2
     */
103 bool sortDescending(Movie* m1, Movie* m2, Attribute attribute);
};
105 /**
 * The method to input the movies into the database from a txt file
107 * @param input - the input stream
 * @param db - the database to be populated
109 * @return the input stream
 */
111 std::istream &operator>>(std::istream &input, MovieDatabase &db);

113
#endif //INC_5015Y_CW2_MOVIEDATABASE_H
```

# MovieDatabase.cpp

```
/**
 * File - MovieDatabase.cpp
 * Author - Luke Marden
 * Date - 21/04/2020
 * Description - A class to handle a database of movie objects
 */
#include <iomanip>
#include <functional>
#include "MovieDatabase.h"

/**
 * Constructor
 */
MovieDatabase::MovieDatabase(){}
/**
 * Destructor
 */
MovieDatabase::~MovieDatabase() {
    for(Movie *movie : this->db) {
        delete movie;
    }
}
/**
 * Makes a sub database based on an attribute and a value
 * @param attribute - the attribute to build the sub database on
 * @param value - the value to match
 * @return The sub database
 */
MovieDatabase* MovieDatabase::subDB(Attribute attribute, std::string value) {
    MovieDatabase* subDB = new MovieDatabase();
    switch (attribute) {
        case NAME:
            for (Movie* movie : this->db) {
                if(movie->getName().find(value) != std::string::npos){
                    subDB->addMovie(movie);
                }
            }
            break;
        case YEAR:
            for (Movie* movie : this->db) {
                if (movie->getYear() == stoi(value)){
                    subDB->addMovie(movie);
                }
            }
            break;
        case CERTIFICATE:
            for (Movie* movie : this->db) {
                if (movie->getCertificate() == value) {
                    subDB->addMovie(movie);
                }
            }
            break;
        case GENRE:
            for (Movie* movie : this->db) {
                if(movie->getGenre().find(value) != std::string::npos){
                    subDB->addMovie(movie);
                }
            }
            break;
        case DURATION:
            for (Movie* movie : this->db) {
```

```
62                  if (movie->getDuration() == stoi(value)){
                        subDB->addMovie(movie);
64                  }
                }
66              break;
            case RATING:
68              for (Movie* movie : this->db) {
                    if (movie->getRating() == stoi(value)){
70                      subDB->addMovie(movie);
                    }
72              }
                break;
74          default: //genre
                for (Movie* movie : this->db) {
76                  if(movie->getGenre().find(value) != std::string::npos){
                        subDB->addMovie(movie);
78                  }
                }
80      }
        return subDB;
82  }
    /**
84   * Sorts the database based on a movies attribute and a direction
     * @param attribute - Movie name etc
86   * @param direction - Ascending/Descending
     */
88  void MovieDatabase::sortDB(Attribute attribute, SortDirection direction) {
        switch(direction){
90          case ASC:
                std::sort(db.begin(), db.end(), [&](Movie* m1, Movie* m2){
92                  return sortAscending(m1, m2, attribute);
                });
94              break;
            case DESC:
96              std::sort(db.begin(), db.end(), [&](Movie* m1, Movie* m2){
                    return sortDescending(m1, m2, attribute);
98              });
                break;
100         default:
                std::sort(db.begin(), db.end(), [&](Movie* m1, Movie* m2){
102                 return sortAscending(m1, m2, attribute);
                });
104     }

106 }
    /**
108      * Overidden ostream method to output every movie in the database
         * @param output - the output stream
110      * @param db - the database
         * @return a series of strings to the console that display all the movies in
             the
112      *          database
         */
114 std::ostream &operator<<(std::ostream& output, const MovieDatabase &db){
        for (Movie *movie : db.db) {
116         std::cout << *movie << std::endl;
        }
118     return output;
    }
120 /**
         * Comparator to see which movie is bigger based on a films attributes
122      * @param m1 - movie 1
         * @param m2 - the movie to compare movie 1 to
```

```
124       * @param attribute - the movies attribute to compare
          * @return true if m2 is bigger than m1
126       */
    bool MovieDatabase::sortAscending(Movie *m1, Movie *m2, Attribute attribute) {
128     switch(attribute) {
            case NAME:
130             return m1->getName() < m2->getName();
            case NAMELENGTH:
132             return m1->getName().length() < m2->getName().length();
            case YEAR:
134             return m1->getYear() < m2->getYear();
            case CERTIFICATE:
136             return m1->getCertificate() < m2->getCertificate();
            case GENRE:
138             return m1->getGenre().length() < m2->getGenre().length();
            case DURATION:
140             return m1->getDuration() < m2->getDuration();
            case RATING:
142             return m1->getRating()<m2->getRating();
            default:
144             return m1->getName() < m2->getName();
        }
146 }
    /**
148       * Comparator to see which movie is bigger based on a films attributes
          * @param m1 - movie 1
150       * @param m2 - the movie to compare movie 1 to
          * @param attribute - the movies attribute to compare
152       * @return true if m1 is bigger than m2
          */
154 bool MovieDatabase::sortDescending(Movie *m1, Movie *m2, Attribute attribute) {
        switch(attribute) {
156         case NAME:
                return m1->getName() > m2->getName();
158         case NAMELENGTH:
                return m1->getName().length() > m2->getName().length();
160         case YEAR:
                return m1->getYear() > m2->getYear();
162         case CERTIFICATE:
                return m1->getCertificate() > m2->getCertificate();
164         case GENRE:
                return m1->getGenre().length() > m2->getGenre().length();
166         case DURATION:
                return m1->getDuration()>m2->getDuration();
168         case RATING:
                return m1->getRating() > m2->getRating();
170         default:
                return m1->getName() > m2->getName();
172     }
    }
174
    /**
176       * The method to input the movies into the database from a txt file
          * @param input - the input stream
178       * @param db - the database to be populated
          * @return the input stream
180       */
    std::istream &operator>>(std::istream &input, MovieDatabase &db){
182     std::string line, data;
        std::ifstream infile("films.txt", std::ifstream::in);
184 //http://www.cplusplus.com/doc/tutorial/files/  template for error checking
        if (infile.is_open()) {
186         while (std::getline(infile, line)) {
```

```
                std::vector<std::string> movieData;
//https://stackoverflow.com/questions/1757065/java-splitting-a-comma-separated-string-but-ignoring-commas-in-quotes
//regex
//https://stackoverflow.com/questions/16749069/c-split-string-by-regex
//general method
                std::regex regex(",(?=(?:[^\\\"]*\\\"[^\\\"]*\\\")*[^\\\"]*$)");
                std::sregex_token_iterator iterator(line.begin(),
                                                    line.end(),
                                                    regex,
                                                    -1);
                std::sregex_token_iterator end;
                for ( ; iterator != end; ++iterator) {
                    std::string string = *iterator;
                    string.erase(remove(string.begin(), string.end(), '"'),
                       string.end());
                    movieData.push_back(string);

                }
                Movie* newMovie = new Movie(movieData[0], stoi(movieData[1]),
                                            movieData[2], movieData[3], stoi(
                                               movieData[4]),
                                            stoi(movieData[5]));

                db.addMovie(newMovie);
            }
            infile.close();
        }
        else{
            std::cout << "File couldn't be found or accessed." << std::endl;
            exit(0);
        }
        return input;

}
```

# main.cpp

```
1  #include <iostream>
   #include <iterator>
3  #include <iomanip>
   #include "Movie.h"
5  #include "MovieDatabase.h"
   int main() {
7      //Populating the database
       MovieDatabase* db = new MovieDatabase();
9      std::cin >> *db;

11     //Task 1
       std::cout << "Task 1: Films in chronological order." << std::endl;
13     db->sortDB(YEAR, ASC);
       std::cout << *db << "\n\n";
15
       //Task 2
17     std::cout << "Task 2: The third longest Film-Noir film." << std::endl;
       MovieDatabase* filmNoirDB = db->subDB(GENRE, "Film-Noir");
19     filmNoirDB->sortDB(DURATION, DESC);
       std::cout << *filmNoirDB->get(2) << "\n\n";
21
       //Task 3
23     std::cout << "Task 3: The eighth most recent UNRATED film." << std::endl;
       MovieDatabase* unratedDB = db->subDB(CERTIFICATE, "UNRATED");
25     unratedDB->sortDB(YEAR, DESC);
       std::cout << *unratedDB->get(7) << "\n\n";
27
       //Task 4
29     std::cout << "Task 4: The Movie with the longest title." << std::endl;
       db->sortDB(NAMELENGTH, DESC);
31     std::cout << *db->get(0) << "\n\n";
       return 0;
33 }
```