COMP 2404 -- Assignment #1

Due: Tuesday, February 4, 2020 at 12:00 pm (noon)

Goal

For this assignment, you will write a C++ program to manage personal movie collections. You will practice writing simple classes in C++, as as well working with dynamically allocated memory.

Learning Outcomes

With this assignment, you will:

- write code with simple C++ classes, and implement a collection class
- work with dynamically allocated memory and pointers
- write and package a program following standard Unix programming conventions

Instructions

1. Modify the `Movie` class:

You will begin with the `Movie` class that we worked on during the lectures. You can find this class in the coding examples posted in *cuLearn*, in section 1.5, program #5 (`S1.5.CtorDtor/p5-Movie-conv`). You will be modifying this class to add some data members and functions, and to remove others.

Modify the `Movie` class as follows:

(1) remove the screenwriter and duration data members, and add a new data member for the year that the movie was made (this will be represented as an integer); modify the constructor accordingly

(2) modify the `print()` function so that it prints the all correct data members (title and year)

(3) remove the conversion constructor, and implement a copy constructor

(4) implement a getter for the `year` data member; you will need this member function in a later step, in order to find the correct position to add a movie to a movie group

2. Implement the `MovieGroup` class

You will create a new `MovieGroup` class that manages a primitive array of movies and provides the required member functions to manipulate those movies. The `MovieGroup` class will contain the following:

(1) a data member that holds the movie collection
   (a) this will be a statically allocated array of `Movie` object **pointers**
   (b) you will define a preprocessor constant for the maximum number of movies; this can be set to a reasonable number such as 64
   (c) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

(2) a data member to track the current number of movies in the array

(3) a constructor that initializes the current number of movies

(4) a destructor that deallocates the dynamically allocated movies contained in the array

(5) a copy constructor that performs a **deep copy** of the movie group; using correct design principles, this function must call the `Movie` copy constructor to create a copy of each `Movie` object

(6) a `print()` member function that prints out every `Movie` object contained in the array; using correct design principles, this function must call the `print()` function of each `Movie` object

(7) a `void add(Movie* m)` member function that adds the given movie `m` to the array in its correct place, in *ascending* (increasing) order by year
   (a) you must **shift** the elements in the array towards the back of the array to make room for the new element in its correct place; **do not** add to the end of the array and sort; **do not** use any sorting function or sorting algorithm
   (b) you must perform all basic error checking

(8) a `void merge(MovieGroup& mg)` member function that takes every movie in the parameter movie group `mg`, makes a copy of the movie (using the `Movie` copy constructor), and adds that movie to the array
   (a) for example: assuming two movie groups `mg1` and `mg2`, if we have the following function call: `mg1.merge(mg2);` then every movie in `mg2` will be copied and the copy added to `mg1`; as a result, `mg1` will contain all of its own original movies, plus copies of all the movies found in `mg2`, all organized in ascending order by year
   (b) you must reuse existing functions everywhere possible, specifically the `add()` function
   (c) you must perform all basic error checking

3. Write the main and initialization functions

Your `main()` function must test your program thoroughly. It will declare two movie groups, and initialize them with *different* movies. It will make a copy of one of the movie groups, and merge this new copy with the other group, so that the copy becomes a "super" group containing all the movies. At the end of the program, it will print all three movie groups.

You will write the `void initMovies(MovieGroup&, MovieGroup&)` global function that initializes two groups of movies. This initialization function will do the following:

(1) dynamically allocate `Movie` objects for at least 28 different movies, **without duplicate information**

(2) initialize the two movies groups with at least 15 movies each

(3) a maximum of **two** movies can be in both groups

(4) the movies must be added to each group in different order of years, so that the movie group's `add()` member function is thoroughly tested

NOTE: If you do not provide data that thoroughly tests your program, you may get **zero** for the functions that cannot be adequately tested.

Alternately, you can use the `initMovies()` function that is posted in *cuLearn* in the `a1-posted.cc` file.

Your `main()` function will do the following:

(1) declare two movie groups, each one associated with a person or fictional character of your choice; for example, here we choose celebrated explorers Calvin and Hobbes

(2) call the `initMovies()` function to initialize the two movie groups

(3) declare the `allMovies` movie group and initialize this group with the content of the first movie group (for example, the Calvin group), by calling the `MovieGroup` copy constructor, which performs a deep copy
   (a) see the coding examples in section 1.5, program #4, for an example of calling the copy constructor using initialization syntax

(4) merge the second movie group (for example, the Hobbes group) into the `allMovies` group, by calling the `allMovies` group's `merge()` member function

(5) print the movies in all three movie groups: the first group (Calvin), the second group (Hobbes), and the `allMovies` group
   (a) the first and second groups should only contain the movies with which they were initialized
   (b) the `allMovies` group should contain all movies in the Calvin group and the Hobbes group combined, all in ascending order by year; note that there will be duplicates if both groups contained the same movie

4. Packaging

Every assignment in this course is required to follow the expected packaging rules for Unix-based systems:

(1) your code must be correctly separated into header and source files, as seen in class

(2) you must provide a Makefile that compiles and links all your code into a working executable

(3) you must provide a README file that contains a preamble (program and revision authors, purpose, list of source/header/data files), as well as compiling and launching instructions

(4) **do not** submit object files, or executables, or duplicate files, or swap files, or supplementary files

5. Test the program

(1) make sure that the data you provide exercises all your functions thoroughly; failure to do this will result in **major deductions**, even if the program appears to be working correctly

(2) check that the movie information is correct when it is printed at the end of the program

(3) make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used; use `valgrind` to check for memory leaks

Constraints

- do not use any classes, containers, or algorithms from the C++ standard template library (STL)
- do not use any global variables or global functions other than the ones explicitly permitted
- do not use structs; use classes instead
- objects must always be passed by reference, never by value
- functions must return data using parameters, not using return values, except for getter functions
- existing functions must be reused everywhere possible
- all basic error checking must be performed
- all dynamically allocated memory must be explicitly deallocated
- your classes must be thoroughly documented in every class definition, as discussed in the course material, section 1.3

Submission

You will submit in *cuLearn*, before the due date and time, the following:

- one **`tar`** or **`zip`** file that includes:
  - all source and header files
  - a Makefile
  - a README file that includes:
    - a preamble (program and revision authors, purpose, list of source/header/data files)
    - compilation and launching instructions

NOTE: Do **not** include object files, executables, swap files, or duplicate files in your submission.

Grading (out of 100)

Marking components:

- 14 marks: correct modifications to `Movie` class
- 58 marks: correct implementation of `MovieGroup` class
  - 7 marks: correct class definition
  - 2 marks: correct implementation of constructor
  - 5 marks: correct implementation of destructor
  - 12 marks: correct implementation of copy constructor
  - 4 marks: correct implementation of `print()` function
  - 12 marks: correct implementation of `add()` function
  - 16 marks: correct implementation of `merge()` function
- 28 marks: correct implementation of `main()` function

Execution requirements:

- all marking components must be called, and they must execute successfully to receive marks
- all data handled must be printed to the screen for marking components to receive marks

Deductions:

- Packaging errors:
  - 10 marks for missing Makefile
  - 5 marks for a missing README
  - 10 marks for consistent failure to correctly separate code into source and header files
  - up to 10 marks for bad style or missing documentation
- Major programming and design errors:
  - 50% of a marking component that uses global variables, or structs
  - 50% of a marking component that consistently fails to use correct design principles
  - 50% of a marking component that uses prohibited library classes or functions
  - up to 100% of a marking component where Constraints listed are not followed
  - up to 10 marks for memory leaks
- Execution errors:
  - 100% of a marking component that cannot be tested because it doesn't compile or execute in VM
  - 100% of a marking component that cannot be tested because the feature is not used in the code
  - 100% of a marking component that cannot be tested because data cannot be printed to the screen
  - 100% of a marking component that cannot be tested because insufficient datafill is provided