COP3330 Programming Assignment 4

DEADLINE: Thursday Mar. 14, 2019

Objective: Upon completion of this program, you should gain experience in working with **dynamic arrays of objects**, as well as working with two classes in a "has-a" relationship (composition). This will also give some extra practice with arrays usage and c-strings/strings.

ABET/SMALC Assessment

This assignment is designated as one of the course assignments being used to assess basic programming skills for ABET/SMALC requirements (see the syllabus for more details). In addition to the normal grading scales, each student's submission will be judged on several aspects on a scale of "Highly Effective", "Effective", or "Ineffective", as specified by ABET/SMALC outcome assessment procedures.
A student's submission that earns at least 85% of the available points will count as "**Highly Effective**", earning between 70-85% of the available points will count as "**Effective**", and getting less than 70% of the points (i.e., a score under 28 points for this assignment) will count as "**Ineffective**".
In order to pass the class you must obtain at least 70% of the points ("Effective") on ONE of the two ABET/SMALC assignments, with this assignment being one of them. For this assignment, that means getting at least 28 points out of the total 40 points.

Overview: You will be writing classes that implement a playlist simulation for a music streaming app. The ***Playlist*** class will contain a dynamic array of ***Song*** objects, and the *Song* class contains several pieces of information about a song. You will need to:

1) Finish the writing of two classes: *Song* and *Playlist*. The full header file for the *Song* class has been provided in a file called ***Song.h***.
2) Write a main program (filename ***menu.cpp***) that creates a single Playlist object and then implements a menu interface to allow interaction with the object.

Program Details and Requirements

1. Using the Song class declaration, write the ***Song.cpp*** file and define all of the member functions that are declared in the file ***Song.h***. (**Do not change *Song.h* in any way. Just take the existing class interface and fill in the function definitions**). Notice that there are only **five** genres of songs in this class: COUNTRY, EDM, POP, ROCK, and R&B. The expected functions' behaviors are described in the comments of this header file.

2. Write a class called Playlist (filenames should be ***Playlist.h*** and ***Playlist.cpp***). A *Playlist* object should contain **at least** the following information: the name of the Playlist, the **total** duration of the Playlist, and a list of songs. There is **no size limit** to the list of songs, so it should be implemented with a dynamically allocated array (this means you will need a dynamic array of *Song* objects. **Note: std::vector is not allowed here in this assignment!**). You can add any public or private functions into the *Playlist* class that you feel are helpful.

3. In addition to the *Playlist* class itself, you will also create a menu program (***menu.cpp***) to manage the list of songs in an interactive way. However, the *Playlist* class will do most of the work, and the *Playlist* member functions will be the interface between the menu program and the Playlist's internal data (name, duration, and list of Songs). The menu program (menu.cpp) only takes charge of iteratively interacting with end-users by showing menus, accepting input from end-users, calling corresponding functions provided by the playlist, and showing the results properly to end-users.

   ### Rules for the *Playlist* class:

   - All member data of class *Playlist* must be **private**.
   - There should be **NO** cin statements inside the *Playlist* class member functions. To ensure the class is more versatile, any **user input** (i.e., keyboard) described in the menu program below should be done in the *menu* program itself. Design the Playlist class interface such that any items of information from outside the class are received through *parameters* of the public member functions.
   - There should be **NO** cout statements in the Song class outside of the `display` function.
   - The list of Songs must be implemented with a dynamically allocated array. There should never be more than 5 unused slots in this array (i.e., the number of allocated spaces may be **at most** 5 slots larger than the number of slots that are actually filled with real data). This means that you will need to ensure that the array allocation **expands** or **shrinks** at appropriate times. Whenever the array is resized, print the following a message (***"** Array being resized to <number of slots> allocated slots"***) that states that the array is being resized, and what the new size is.
     Example: *"** Array being resized to 10 allocated slots".*
   - Since dynamic allocation is being used inside the *Playlist* class, an appropriate destructor must be defined, to clean up memory. **The class must not allow any memory leaks!**
   - You must use the const qualifier in all appropriate places where it makes sense (i.e., const member functions and const parameters).

4. The main program (***menu.cpp***) you write should create a single *Playlist* object and then implement a menu interface to allow interaction with the object. The program should begin by asking the user to input a name for the *Playlist*. This name should be stored in the *Playlist* object. You may assume user entry will be a string no longer than 20 characters. The name of the playlist will be updated to the value entered by the user, unless the new name is empty or made up of only white spaces. In the latter case display an appropriate message informing the user that the name was invalid and ask the user to enter the data again (example**:** "Invalid playlist name. Please re-enter:").

   Your main program should implement the following menu loop (any single letter options should work with **BOTH** lower and upper case inputs):

   A: Add a song to the playlist
   F: Find a song in the playlist
   R: Rename playlist
   S: Remove a song
   D: Display the playlist
   G: Genre summary

M: Show this Menu

Q: Quit/exit the program

Behavior of menu selections:

All user inputs described in the menu options below should be done in the menu program (not inside the *Song* or *Playlist* classes). The input should then be sent to the *Playlist* class -- the *Playlist* class member functions should do most of the actual work, since they will have access to the list of songs. For all user inputs (keyboard), assume the following:

- A song **title** and **artist** will always be input as strings (c-style strings), of maximum lengths 40 and 25, respectively. You may assume that the user will ***NOT*** enter more characters than the stated limits for these inputs.
- A playlist's **name** will always be input as a string (c-style strings) of maximum length 20. You may assume that the user will ***NOT*** enter more characters than the stated limit for this input. If the input is empty or made up of spaces only, display an error message and ask the user to enter the data again (example error message: "Invalid name. Please re-enter: ").
- When asking for the **genre**, user entry should always be a single character. The correct values are C, E, P, R, and B; for country, EDM, pop, rock, and R&B, respectively. Uppercase and lowercase inputs should both be accepted. Whenever the user enters any other character for the genre, this is an error -- print an error message and prompt the user to enter the data again (example error message: "Invalid genre entry. Please re-enter: ").
- User input of a song's duration should be in seconds, and should be a positive **integer**. Whenever the user enters a number that is not positive, it is considered an error -- so an error message should be printed and the user should be prompted to re-enter the duration (example error message: "Must enter a positive duration. Please re-enter: "). Note that zero is NOT a positive number.
- User input of menu options are letters, and both upper and lower case entries should be allowed.

A: This menu option should allow the adding of a Song to the Playlist. The user will need to type in the song's information. Prompt and allow the user to enter the information in the following order: **title, artist, genre, duration**. The information should be sent to the Playlist object and stored in its list of songs.

F: This option should allow the user to search for a Song in the Playlist by **title** or by **artist**. When this option is selected, ask the user to enter a search string (may assume user entry will be a string no longer than 40 characters). If the search string matches **exactly** a Song title, display the information for that Song (output format is described in the *display* function of the Song class). If the search string matches an artist in the Playlist, display the information for **all** songs by that artist. Note that a search string could match several songs and several artists. In that case, print the information for all the matches that are found. If no matching songs are found, display an appropriate message informing the user. (example: No songs found).

R: This option should allow the user to enter a new name for the Playlist. When this option is selected, ask the user to enter a new name for the playlist as a string (may assume user entry will be a

string no longer than 20 characters). The name of the playlist will be updated to the value entered by the user, unless the new name is empty or made up of only white spaces. In the latter case no update will be made and an appropriate message informing the user that the name was invalid and no update was performed should be displayed (example**:** "Invalid playlist name. Playlist name was not updated").

S: This option should allow the user to remove a song from the playlist. When this option is selected, ask the user to type in the title of the song and remove this song from the playlist. If there is no such title, inform the user and return to the menu (example: "No songs removed from *PlaylistName*"). If there is more than one Song with the same title, delete ALL of the songs with that title from the playlist.

D: This option should display the entire Playlist, one Song per line, in an organized manner (as mentioned in the description of the function *display* in the Song class). Each line should contain one Song's information, as described in the ***Song.h*** file. After this, also display the total number of songs in the playlist, as well as the total duration of the playlist. The duration should be printed as minutes and seconds notation (m:ss).
If the playlist is empty, an appropriate message should be displayed instead (example: "The playlist <Playlist name> is Empty").

G: This option should list the Playlist's songs for one specific genre. When this option is selected, ask the user to input a genre. For the genre selected, print out the Songs in the Playlist, as in the **Display (D)** option, but **only for the Songs matching the selected genre**. After this, also display the number of songs in this genre, as well as the total duration (the sum of the individual durations) of the songs in this genre. The total duration should be printed as minutes and seconds notation (m:ss). If the playlist is empty, an appropriate message should be displayed instead (example: "The playlist <Playlist name> is empty").

M: Re-display the menu.

Q: Exit the menu program. Upon exiting, print out the playlist and a goodbye message and the length of the playlist (example: Exiting program. Your playlist was m:ss long).

Note: the interface for Song.h indicates that all the strings are represented as c-style strings. Please do NOT change the interface! However, inside the implementation of Song and Playlist, if you want to use C++string (don't forget to #include <string>), that is still allowed. That is, you may need to transform between c-style strings and modern c++ strings in your implementation. On the other hand, you can also consistently use c-style strings in all your implementations.

General Requirements

1. All member data of the class must be private.
2. Do NOT use std::vector in this assignment. Do not use language or library features, third-party libraries, or OS-specific API calls.
3. Adhere to the good programming practices discussed in class (e.g., no global variables, other than constants or enumerations, **DO NOT** #include .cpp files, properly document your code).
4. Your ***.h*** files should contain the class declaration only. The ***.cpp*** files should contain the member function definitions.

5. You only need to do error-checking that is specified in the descriptions above. If something is not specified (e.g. user entering a letter where a number is expected), you may assume that part of the input will not be evaluated;
6. When you write source code, it should be readable and well-documented.

Extra Credit

Write a function called ***sort(char)***, and add in a menu option (using the letter 'O') for sorting the Playlist. When this menu option is chosen, ask the user whether they want to sort by artist or title (enter 'A' or 'T', allowing both uppercase and lowercase entries). Then, sort the Playlist songs in **ascending** alphanumeric order, on the appropriate field (artist or title).

Sample Executable

We provide a reference executable for this assignment on the course website. You can run it from linprog.cs.fsu.edu with the command:

```
./mymusicapp
```

Please note that this executable is compiled in linprog.cs.fsu.edu, so it may not run on Windows or Mac platforms.

You should match the output of the reference executable, including formatting.

Submission

Program submissions should be done through Canvas. Do not send program submissions through e-mail -- e-mail attachments will not be accepted as valid submissions!

For Assignment 4, submit a compressed package (named **proj4.tar.gz**) including the following files

- `Song.cpp`
- `Playlist.h`
- `Playlist.cpp`
- `menu.cpp`
- `makefile` (the final executable is named `proj4`, and the file containing main() function is named `menu.cpp`)
- `readme` (optional)

Do NOT submit the **Song.h** file (or any other main program you might create for testing purposes). Make sure your filenames are these EXACT names.