CSCI 1300 CS1: Starting Computing
Naidu/Correll/Yeh/Hoefer Fall 2021
Project 2
Due: Tuesday, Nov 2nd, 2021

---

0. Table of Contents

---

1. Objectives

- Use filestream objects to read data from text files
- Create objects
- Array operations: arrays of objects

---

2. Interview Grading

Sign up for an interview grading slot on Canvas. The interviews for this project will be hosted between **November 3rd** and **Nov 10th**. If you don't sign-up and you miss your interview, then **no points will be awarded** for the project.
The schedulers for interview grading will be available on Canvas before the deadline of this project.

- Please come to the interview a few minutes ahead of time so that you can be ready and avoid any technical difficulties.
- Have your code pulled up and ready on VS code before you start your interview.
- Please remember to turn on your camera during the interview.

Special Requirements for Homework 7 and Project 2 (These components are not allowed)

- **global variables,**
- **vectors,**
- **references,**
- **pointers and**
- **stringstream.**

---

3. Submission Requirements

All three steps must be fully completed by the submission deadline for your project to be graded.

1. ***Work on questions on your VS Code workspace:*** You need to write your code in VS Code workspace to solve questions ***and*** test your code in your VS Code workspace before submitting it to Canvas. (Create a directory called **Project2** and place all your file(s) for this assignment in this directory to keep your workspace organized). You will need to develop your code in multiple files: the header (.h file with class definition), the implementation (.cpp file with implementations of class member functions) and the Driver file (.cpp file with the main() function)

2. ***Submit to the Canvas Coderunner:*** Head over to Canvas to **Project 2 Coderunner**. You will find one programming quiz question for each problem in the assignment. Submit your

solution for each problem by pasting the required class definition from your header file, the class implementations from the cpp file, and any required driver functions in the box, and press the Check button. You will see a report on how your solution passed the tests, and the resulting score for the first problem. You can modify your code and re-submit (press *Check* again) as many times as you need to, up until the assignment due date. Continue with the rest of the problems.

3. ***Submit a .zip file to Canvas:*** After you have completed all 11 questions and tested them on CodeRunner, zip all your solution files from VS Code (all .cpp and .h files) and submit the zip file through the **Project 2 zip submission** link on Canvas. NOTE: *Your program must compile and run, so make sure you include your driver files with int main() for the problems*.

For each question, check out below what files need to be submitted, and what is expected from each file!

---

4. Problem Set

Note: To stay on track for the week, we recommend to finish/make considerable progress on problems 1-6 by Thursday. Students are encouraged to read through the writeup and start on the first few problems before recitations so that they can bring their questions to recitation.

In this project, you have been contracted by an online music streaming service that contains a collection of songs from an eclectic bunch of music artists. Your client would like to begin doing data analysis on their listener base, and would like to create a software program for managing a database of songs and listeners. Your client currently has information about both Songs and Listeners typed into simple text files. Your job is to create a C++ software package that provides necessary functionality for manipulating both Song and Listener objects, and providing simple data analysis features.

Problem 1 (1 point): Song Class - *this is identical to Problem 3 from Homework 7*

Create a `Song` class, with separate interface file (`Song.h`) and implementation file (`Song.cpp`), comprised of the following attributes:

Data members (private):	
string: `title`	Title of the song

string: `artist`	Artist of the song
string: `genre`	Genre the song falls under
Member functions (public):	
Default constructor	Sets `title, artist,` and `genre` to empty strings
Parameterized constructor	Takes three arguments for initializing `title, artist,` and `genre,` in this order
`getTitle()`	Returns `title` as a string
`setTitle(string)`	(void) Assigns `title` the value of the input string
`getArtist()`	Returns `artist` as a string
`setArtist(string)`	(void) Assigns `artist` the value of the input string
`getGenre()`	Returns `genre` as a string
`setGenre(string)`	(void) Assigns `genre` the value of the input string

In the **main()** function of your driver file songDriver.cpp, the test cases should include the creation of class objects with both the default and parameterized constructors. You must also test each of the getter and setter member functions by creating and manipulating class objects and displaying output to verify that things are working properly. Each member function of the class should be called at least once to test it. Refer to stateDriver.cpp to see how to appropriately test your class in this way.

The zip submission should have three files for this problem: **Song.h**, **Song.cpp**, and a driver called **songDriver.cpp**, with a **main()** function to test your member functions. For Coderunner, paste your Song class and its implementation only (the contents of Song.h and Song.cpp). Do not include the `main()` function, nor header statements (`#include <iostream>`, `using namespace std` or `#include "Song.h"`).

Problem 2 (1 point): readSongs - *this is identical to Problem 4 from Homework 7*

The **readSongs** function will fill an array of `Song` objects with `title, artist` and `genre` information. This function should:

- Have four parameters in this order:
  - **string** `fileName`: the name of the file to be read.
  - **array** `songs`: array of **Song** objects.

    - **int** `numSongsStored`: the number of songs currently stored in the array. You can always assume this is the correct number of actual elements in the arrays.
    - **int** `songArrSize`: capacity of the `song` array. You should set the default value for this parameter to 50.
- Use `ifstream` and `getline` to read data from the file, making an instance of the Song object for each line, and placing it into the `songs` array.
- You can use the `split()` function from Problem 5 in Homework 5, with comma (‘,’) as the delimiter. Do not include your version of the split function in the answer box, we will include it, so you can use it and call it if you wish. A call to the split function will look like this:

```
string arr[50]; // Declare an array that will store strings from
line
getline(myFile, line); // Get a line from a file
split(line, ',', arr, 50); // Split the line by commas
```

The array `arr` will contain the substrings that resulted from parsing the string `line`. For reference, each entry in the test files is formatted as "`title,artist,genre`", without the quotation marks.

- When you copy your code to the CodeRunner, make sure you put in the Answer Box your **Song** class, **readSongs()** function
- Empty lines should not be added to the arrays.
- The function should return the following values depending on cases:
    - Return the total number of `songs` in the system, as an integer.
    - When `numSongsStored` is equal to the `songArrSize`, return -2.
    - When the file is not opened successfully, return -1.
    - The priority of the return code -2 is higher than -1, i.e., in cases when `numSongsStored` is equal to the `songArrSize` and the file cannot be opened, the function should return -2.
    - When `numSongsStored` is smaller than `songArrSize`, keep the existing elements in the array `songs`, then read data from the file and add (append) the data to the array. The number of songs stored in the array cannot exceed the `songArrSize`.
- Empty lines should not be added to the arrays

Example 1: The `songs` array is empty, so for this example `numSongsStored` is zero and the function returns 2.

fileName.txt	`Bennie and the Jets,Elton John,Pop` `Lean on Me,Bill Withers,Rhythm and Blues`
Function call	`Song songs[10] = {};` `readSongs("fileName.txt",songs, 0, 10);`
Return value	`2`

Testing the data member `title`	`// Code to print the values`
	`cout<<songs[0].getTitle()<<endl;`
	`cout<<songs[1].getTitle()<<endl;`
	`// Expected Output`
	`Bennie and the Jets`
	`Lean on Me`

Testing the data member `artist`	`// Code to print the values`
	`cout<<songs[0].getArtist()<<endl;`
	`cout<<songs[1].getArtist()<<endl;`
	`// Expected Output`
	`Elton John`
	`Bill Withers`

Testing the data member `genre`	`// Code to print the values`
	`cout<<songs[0].getGenre()<<endl;`
	`cout<<songs[1].getGenre()<<endl;`
	`// Expected Output`
	`Pop`
	`Rhythm and Blues`

Example 2: The `songs` array has one `song`, so for this example `numSongsStored` is 1 and the function returns 4.

fileName.txt	`The Long and Winding Road,The Beatles,Rock`
	`Penny Lane,The Beatles,Rock`
	`Patches,Clarence Carter,Soul`

Function call	`Song songs[10] = {};`
	`songs[0].setTitle("name1");`
	`songs[0].setArtist("artist1");`
	`songs[0].setGenre("genre1");`
	`readSongs("fileName.txt",songs, 1, 10);`

Return value	`4`

Testing the data member `title`	`// Code to print the values`
	`cout<<songs[0].getTitle()<<endl;`
	`cout<<songs[1].getTitle()<<endl;`
	`cout<<songs[2].getTitle()<<endl;`
	`cout<<songs[3].getTitle()<<endl;`
	`// Expected Output`
	`name1`
	`The Long and Winding Road`
	`Penny Lane`
	`Patches`

Testing the data member `artist`	`// Code to print the values`
	`cout<<songs[0].getArtist()<<endl;`
	`cout<<songs[1].getArtist()<<endl;`
	`cout<<songs[2].getArtist()<<endl;`
	`cout<<songs[3].getArtist()<<endl;`
	`// Expected Output`
	`Artist1`

	`The Beatles`
	`The Beatles`
	`Clarence Carter`

Testing the data member `genre`	`// Code to print the values`
	`cout<<songs[0].getGenre()<<endl;`
	`cout<<songs[1].getGenre()<<endl;`
	`cout<<songs[2].getGenre()<<endl;`
	`cout<<songs[3].getGenre()<<endl;`
	`// Expected Output`
	`genre1`
	`Rock`
	`Rock`
	`Soul`

Example 3: The `products array is already full`, so for this the function returns -2

Function call	`Song songs[2] = {};`
	`songs[0].setTitle("title1");`
	`songs[0].setArtist("artist1");`
	`songs[0].setGenre("genre1");`
	`songs[1].setTitle("title1");`
	`songs[1].setArtist("artist1");`
	`songs[1].setGenre("genre1");`
	`readSongs("minitest1.txt", songs, 2, 2);`

Return value	`-2`

Example 4: The `file does not exist`, so for this the function returns -1

Function call	`Song songs[10] = {};` `readSongs("unknown.txt",songs, 0, 10);`
Return value	`-1`

The zip submission should have three files for this problem: **Song.h**, **Song.cpp** and a driver program called **readSongsDriver.cpp**, with your **readSongs()** function and a **main()** function to test your **readSongs()** function. The examples above show the appropriate way to do that.

For **Coderunner**, paste your **Song class** (both the header and implementation)**, and your readSongs function**, not the entire program. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 3 (1 point): printAllSongs - *this is identical to Problem 5 from Homework 7*

Write a **printAllSongs** function to display the contents of your music library. This function should:

- Have two parameters in this order:
  - **array** `songs`: array of **Song** objects.
  - **int** `numsongs`: number of songs in the array (*Note: this value might be less than the capacity of 50*)
- This function does **not** return anything
- If the number of songs is 0 or less than 0, print `"No songs are stored"`
- Otherwise, print `"Here is a list of songs"` and then each song in a new line using the following format (without the quotes): "<title> is by <artist>"

Note: In the test case, you can always assume that the number of songs matches the number of elements in the `songs` array.

Example output

`Here is a list of songs`
`Love Me Do is by The Beatles`
`In the Rain is by The Dramatics`
`Crocodile Rock is by Elton John`
`I Want to Hold Your Hand is by The Beatles`
`...`

The zip submission should have three files for this problem: **Song.h**, **Song.cpp** and a driver program called **printAllSongsDriver.cpp**, with your **printAllSongs()** function and a **main()** function to test your **printAllSongs()** function.

For **Coderunner**, paste your **Song class** (both the header and implementation)**, and your printAllSongs function**, not the entire program. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 4 (5 points): countGenre

Write a standalone **countGenre** function to count all songs of a particular genre. This function essentially filters songs in your songs array by genre and returns the count of songs that match the given genre. This function should:

- Have 3 parameters in this order:
  - **string** `genre`: A string to filter songs that match this Genre (Rock, Pop etc).

        - Please remember that it should be **case insensitive** so for example "rock" should match with "Rock".
            - **Hint**: You may create a helper function which converts the following strings to either uppercase or lowercase. For example, create a helper function called toLower; the purpose of this function is to take in a string and return the lowercase of that string. The idea is to use this function on strings during case insensitive comparisons so that whichever case they may be we can tell if they are the same word or not.
    - **array** `songs`**:** array of **Song** objects.
    - **int** `numSongsStored`: number of Songs in the array (*Note: this value might be less than the capacity of 50*)
- **Return int:** The count of songs that match the given Genre. Return 0 if no songs match (given genre is not found) or if the array is empty or if the number of songs are invalid (like a negative number).

Example 1: There exists at least one song in the array of Song objects, that has the `genre` we are looking for

songs1.txt	`Goodbye Yellow Brick Road,Elton John,Pop`
	`Turn Back the Hands of Time,Tyrone Davis,Rhythm and Blues`
	`The Payback,James Brown,Rhythm and Blues`
	`I Want to Hold Your Hand,The Beatles,Rock`

Function call	`Song song[50];`
	`int numSongsStored = 0;`
	`int songArrSize = 50;`
	`int i = readSongs("songs1.txt",song, numSongsStored, songArrSize);`
	`// update numSongsStored based on i`
	`...`
	`cout << countGenre("Rock", song, numSongsStored);`

Return value	`1`

Example 2: There are no songs in the array of Song objects that have the `genre` we are looking for or the song file is invalid.

noGenre.txt	`Goodbye Yellow Brick Road,Elton John,Pop` `Bennie and the Jets,Elton John,Pop`

	`Crocodile Rock,Elton John,Pop`
	`Border Song,Elton John,Pop`
	`Love on a Two Way Street,The Moments,Soul`

Function call	`Song song[50];`
	`int numSongsStored = 0;`
	`int songArrSize = 50;`
	`int i = readSongs("noGenre.txt",song,`
	`numSongsStored, songArrSize);`
	`// update numSongsStored based on i`
	`...`
	`cout << countGenre("Metal", song,`
	`numSongsStored);`

Return value	`0`

The zip submission should have three files for this problem: **Song.h**, **Song.cpp** and a driver program called **countGenreDriver.cpp** to test your member functions and the countGenre function. For **Coderunner**, paste **only your Song class implementation, countGenre function**, and any helper functions that you have created to assist you. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 5 (7 points): frequentGenreSongs

Write a standalone **frequentGenreSongs** function to count all the songs of the most frequent genre. **Here most frequent genre refers to the genre with the most songs in the song array**. This function essentially finds the most frequent genre from the song array and returns the count of the songs in that genre. This function should:

- Have 2 parameters in this order:
    - **array** `songs`: array of **Song** objects.
    - **int** `numSongsStored`: number of songs in the array (*Note: this value might be less than the capacity of 50*)
- **Return int:** The count of songs of the most frequent genre. If more than one genre has the same highest frequency (i.e in case of a tie) then return the count of songs from either one of them. If no songs are found then return 0. **(you may use the countGenre function from problem 4)**

The function should return the following:

- Case 1: If there is a clear majority in the genre from the list of the songs then *return* the count of songs of the most frequent genre.

- Case 2: If there is a tie (i.e there are more than one genre with the same frequency) then return the count of songs from either one of them.
- Case 3: If no songs are found or if the song array is empty or if the value of number of songs stored is invalid *return* 0.

Note: In the test case, you can always assume that the number of songs matches the number of elements in the `songs` array.

HINT: Create two new arrays of equal length (assume length to be at least as big as the song array), one of the arrays is used to store the unique genres that are in the song array and the other to store their corresponding frequency. Essentially both arrays are linked over the same index.

For example:

string Arr1[3] = {“Pop”, “Rock”, “Rap”};
int Arr2[3] = {2, 3, 1};

Here Arr1[0] has the genre Pop and its corresponding frequency of occurrence 2 stored in Arr2[0]. So the index can be used as a connection or a map to associate the values between the two arrays. You will then find the genre which has the highest frequency and print all the songs that belong to that genre. In this situation the genre is Rock has a frequency of 3 and you will print all the songs that belong to the Rock Genre.

Example 1: There exists one genre in the array of Song objects, that has the highest frequency.

songs1.txt	`Goodbye Yellow Brick Road,Elton John,Pop`
	`Turn Back the Hands of Time,Tyrone Davis,Rhythm and Blues`
	`The Payback,James Brown,Rhythm and Blues`
	`I Want to Hold Your Hand,The Beatles,Rock`

Function call	`Song songs[50];`
	`int numSongsStored = 0;`
	`int songsArrSize = 50;`
	`numSongsStored = readSongs("songs1.txt", songs, numSongsStored, songsArrSize);`
	
	`cout<<frequentGenreSongs(songs,numSongsStored)<<endl;`

Return	`2`

value	

Example 2: There exists more than one genre in the array of Song objects, that has the highest frequency.

songs2.txt	Goodbye Yellow Brick Road,Elton John,Pop The Payback,James Brown,Rhythm and Blues I Want to Hold Your Hand,The Beatles,Rock

Function call	Song songs[50];
	int numSongsStored = 0;
	int songsArrSize = 50;
	numSongsStored = readSongs("songs2.txt", songs, numSongsStored, songsArrSize);
	
	cout<<frequentGenreSongs(songs,numSongsStored)<<endl;

Return value	1

Example 3: There are no songs in the song array or if file does not exist

songs3.txt	

Function call	Song songs[50];
	int numSongsStored = 0;
	int songsArrSize = 50;
	numSongsStored = readSongs("songs3.txt", songs, numSongsStored, songsArrSize);
	
	cout<<frequentGenreSongs(songs,numSongsStored)<<endl;

Return value	0

The zip submission should have three files for this problem: **Song.h**, **Song.cpp** and a driver program called **frequentGenreSongsDriver.cpp** to test your member functions and the searchNameGenre function. For **Coderunner**, paste **only your Song class implementation,**

countGenre (function if used) and **frequentGenreSongs** function, not the entire program. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 6 (8 points): Listener Class

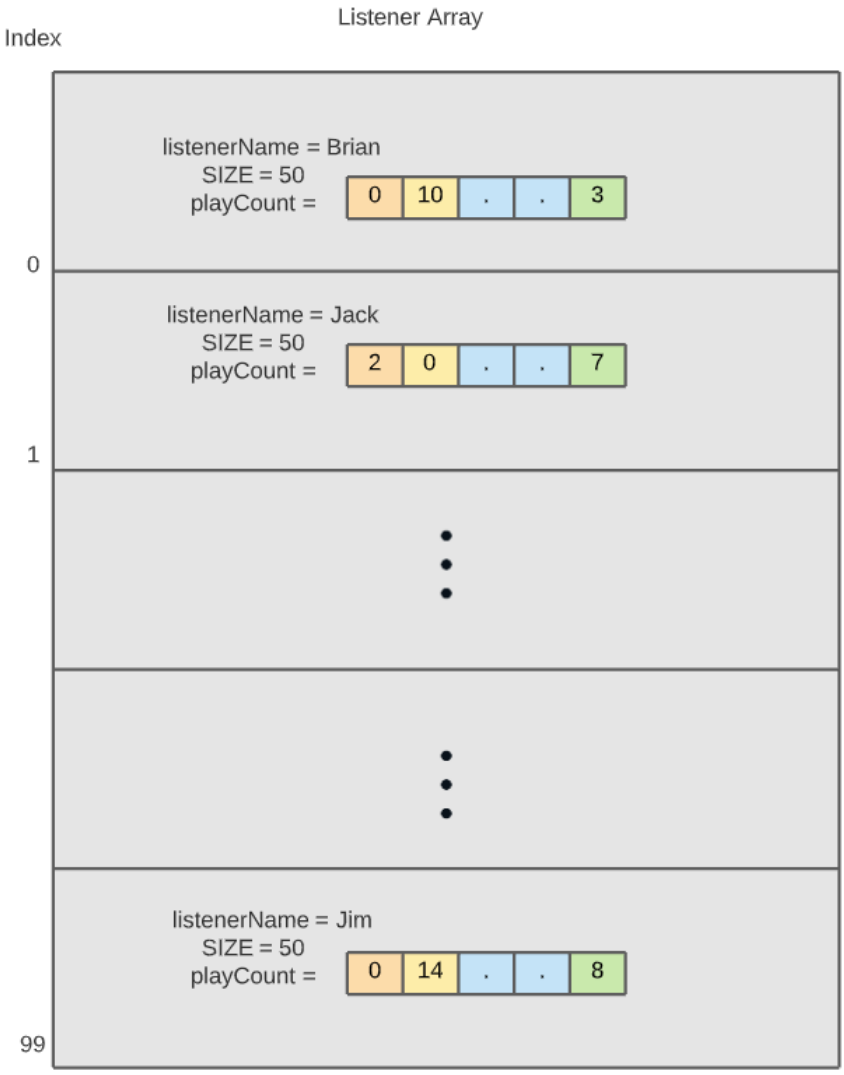
Create a `Listener` class, with separate interface (`Listener.h`) and implementation (`Listener.cpp`) with class functions, comprised of the following attributes:

Data members (private):	
string: `listenerName`	The name of the listener
int array: `playCount`	Size of this array should be `size`. The values stored in this array will represent the count of plays for each song (i.e the number of listens of each song) listened by the listener. Each index of the array corresponds to the same index in a Song array, so the number of plays for song `i` can be found in `playCount[i]`.
static const int: `size`	The capacity of the `playCount` array (50). Constant
Member functions (public):	
Default constructor	Sets `listenerName` to an empty string and all the elements in the `playCount` array to the value 0.
Parameterized constructor	Takes a string for initializing listenerName, an array of integers for initializing `playCount`, and an integer with the size of the array passed in. Only save the first size elements of the input array, disregard anything larger. If the input array is smaller than size, then fill the rest of the array with zeros.
`getListenerName()`	Returns `listenerName` as a string
`setListenerName(string)`	(void) Assigns `listenerName` the value of the input string
`getPlayCountAt(int)`	Parameter: `int index`. Returns the count of plays of a song (i.e the number of listens of a song) stored at the specified index. If the `index` is larger than the last index in the playCount array, or less than 0, returns -1.

`setPlayCountAt(int,int)`	Parameters: `int index, int value`. Sets the count of number of plays of songs (i.e the number of listens of a song) to `value` at the specified `index`, if `index` is within the bounds of the array and `value` is greater than 0. Returns a boolean value, `true` if the number of plays was successfully updated and `false` otherwise. To be clear - one cannot set the playCount array equal to zero using this function.
`totalPlayCount()`	Calculates and returns the sum of all the number of plays of all songs (i.e the number of listens of all songs) , returns an `int`
`getNumUniqueSongs()`	Calculates and returns the number of *unique* songs listened to by the listener; unique here refers to songs which have at least one play (value >= 1 in the playCount array)
`getSize()`	Returns `size` as an integer

In your `main()` function, the test cases should include the creation of class objects with both the default and parameterized constructors. You must also test each of the getter and setter member functions by creating and manipulating class objects and displaying output to verify that things are working properly. Refer to stateDriver.cpp to see how to appropriately test your class in this way.

The zip submission should have three files for this problem: **Listener.h**, **Listener.cpp**, and a driver called **listenerDriver.cpp**, with a `main()` function to test your member functions. For **Coderunner**, paste your **Listener class and its implementation** (the contents of Listener.h and Listener.cpp). Do not include the `main()` function. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).



Problem 7 (15 points): readListenerInfo

Write a standalone function **readListenerInfo** that loads the listener logs by reading the `playlist.txt` file. The first value of each line in `playlist.txt` is the listenerName. Each name is followed by a list of the number of plays for each song in the listener's playlist.

For example, let us say there are in total 3 songs. The `playlist.txt` file would be of the format:

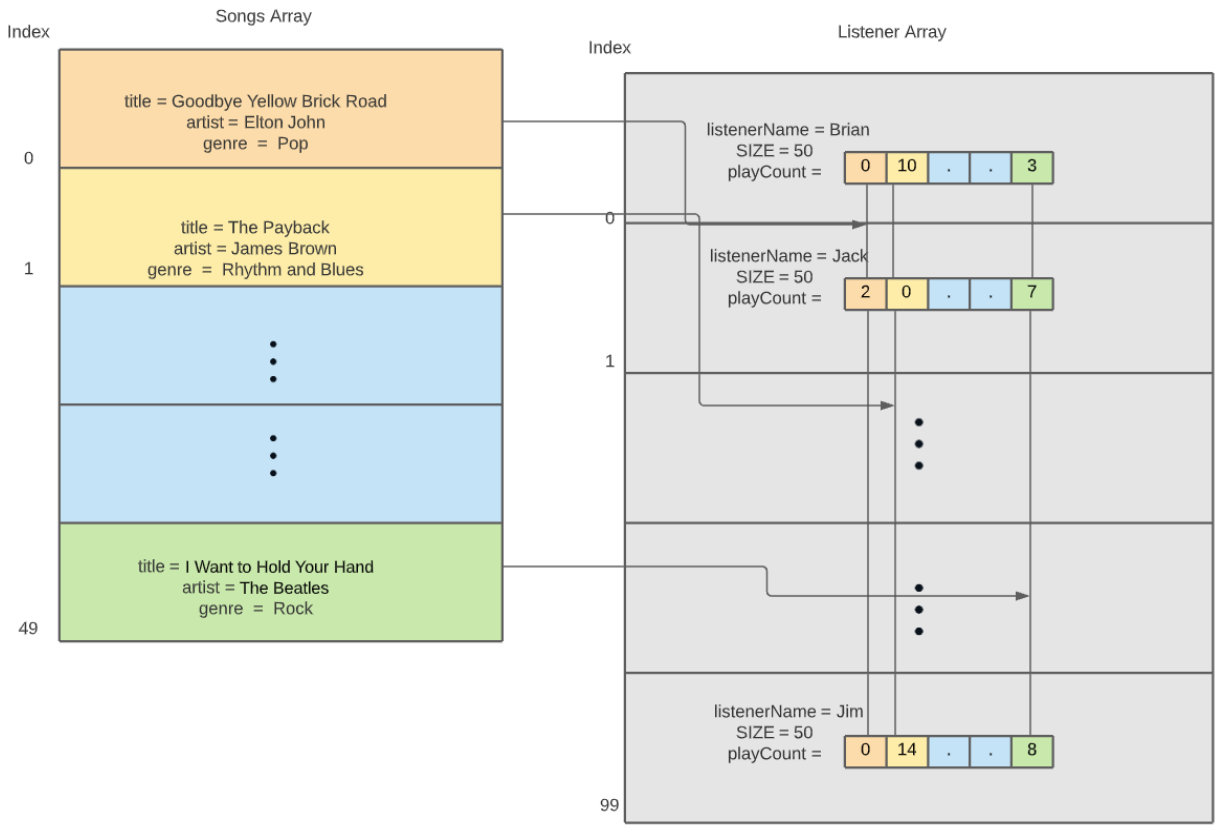
```
Al,0,0,1
John,0,0,2
Sleve,1,0,1
Onson,2,1,0
```

```
. . .
```

In this case, Al has played song 0 times of song 0, 0 times of song 1, and 1 time of song 2.

This function should:

- Accept five parameters in this order:
  - **string** `filename`: the name of the file to be read
  - **array** `listeners`: array of **Listener** objects
  - **int** `numListenersStored`: number of listeners currently stored in the array
  - **int** `listenerArrSize`: capacity of the `listeners` arrays. The default value for this data member is 100.
  - **int** `maxCol`: maximum number of columns. The default value for this data member is 51. (each line in the file contains 1 name and 50 integers for play counts. *Note:* if your maxCols is less than the actual number of columns in the file you may run into a segmentation error if not handled properly)
- Use `ifstream` and `getline` to read data from the file, making an instance of a `Listener` object for each line, and placing it in the `listeners` array.
- Assume that all listeners in the input file (`filename`) are distinct.
- **Hint**: You can use the `split()` - function from Homework 5, with comma (“,”) as the delimiter. When you copy your code in the Answer Box on Coderunner, make sure you copy the **Listeners** class and **readListenerInfo()** function but not **split()** (which is provided for you).
- You can use `stoi` function part of the std library to convert each number of plays value (a string, as read from the text file) into an integer value.
- Empty lines should not be added to the arrays.
- The function should return the following values depending on cases:
  - Case1: If `numListenersStored` is greater than or equal to the `listenerArrSize`, return -2.
  - Case2: If the file cannot be opened, return -1.
  - Case 3: If file exists but it is empty return 0
  - Case4: If `numListenersStored` is smaller than the `size` of `listeners` array, keep the existing elements in `listeners` array, then read data from file and add (append) the data to the arrays. The number of listeners stored in the arrays cannot exceed the size of the `listeners` array. Return the total number of listeners in the array, as an integer.
  - Your function must check these cases in the order specified above.



Example 1: The `listeners` array is empty, so `it can read everything in the text file`.

playlist.txt	`Al,0,0,1`
	`John,5,0,2`
	`Sleve,1,0,1`
	`Onson,2,1,0`

Function call	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`readListenerInfo("playlist.txt", listener, numListenersStored, listenerArrSize, 50);`

Return value	`4`

Testing the data member `listener`	`// Code to print the values`
	`for(int i = 0; i <numListenersStored; i++){`
	`cout<<listener[i].getListenerName()<<endl;`
	`for (int j = 0; j<numSongsStored; j++)`
	`cout<<listener[i].getPlayCountAt(j)<<endl;`

	}
	
	// Expected Output
	Al
	0
	0
	1
	John
	5
	0
	2
	Sleve
	1
	0
	1
	Onson
	2
	1
	0

Example 2: The `listeners` array is empty since a bad file which can't be opened is given. *This is not an empty file rather a file that does not exist.*

Playlist.txt	

Function call	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`readListenerInfo("Playlist.txt", listener, numListenersStored, listenerArrSize, 50);`

Return value	-1

Testing the data member `listenerName`	// Code to print the values
	cout<<listeners[0].getListenerName()<<endl;
	cout<<listeners[1].getListenerName()<<endl;
	.
	.
	// Expected Output
	“”
	“”
	.

	.

Testing the data member	// Code to print the values
	cout<<listeners[0].getPlayCountAt(0)<<endl;
	cout<<listeners[0].getPlayCountAt(1)<<endl;
	cout<<listeners[0].getPlayCountAt(2)<<endl;
	
	.
	.
	
	// Expected Output
	0
	0
	0
	
	.
	.

Example 3: The `listeners` array is already full, so `readListenerInfo` returns -2.

morePlaylist.txt	alpha,0,1,2,3,4 beta,0,1,2,3,4 gamma,0,1,2,3,4 delta,0,1,2,3,4

Function call	int numListenersStored = 2;
	int listenerArrSize = 2;
	Listener listener[listenerArrSize];
	readListenerInfo("morePlaylist.txt", listener,
	numListenersStored, listenerArrSize, 50);

Return value	-2

Example 4: There is already 1 listener in the `listeners` array, so the value of `numListenersStored` is 1. However, the array `size` is only two, so only the first line of the file is stored and the function returns the number of listeners in the array.

playlist.txt	stroustrup,0,4,5 gosling,2,2,3 rossum,5,5,5
Function calls	const int listenerArrSize = 2; Listener listeners[listenerArrSize];

	`listeners[0].setListenerName("ritchie");`
	`listeners[0].setSongsAt(0,0);`
	`listeners[0].setSongsAt(1,1);`
	`listeners[0].setSongsAt(2,2);`
	`int numListenersStored = 1;`
	
	`readListenerInfo("playlist.txt",listeners,`
	`numListenersStored, listenerArrSize, 50);`

Return value	`2`

Example 5: File exists but it is an empty file

emptyFile.txt	

Function calls	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`readListenerInfo("Playlist.txt", listener,`
	`numListenersStored, listenerArrSize, 50);`

Return value	`0`

The zip submission should have three files for this problem: **Listener.h**, **Listener.cpp**, and a driver called **readListenerInfoDriver.cpp**, with a `main()` function to test your member functions. For **CodeRunner**, paste your Listener class and its implementation (the contents of **Listener.h** and **Listener.cpp**), and your **readListenerInfo** function. Do not include the `main()` function. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 8 (5 points): getSongPlayCount

We now have a list of songs (in the form of an array of `Song` objects) and a list of listeners and the number of plays of each song they have listened to (in the form of an array of `Listener` objects).

Write a standalone function that, given a listener's name and a song's name, returns the total number of plays of that song played by that listener.

- Your function **MUST** be named **getSongPlayCount**.

- Your function should take 6 parameters in the following order:
    - **string**: listener name
    - **string**: song name
    - **Array of Listener objects**: `listeners`
    - **Array of Song objects**: `songs`
    - **int**: number of listeners currently stored in the `listeners` array
    - **int**: number of songs currently stored the `songs` array
- The listener name and song name search should be case insensitive. For example, “Ben, “ben” and “BEN” are one and the same.
        - **Hint**: (You might need to create a helper function which helps you convert strings to either of the cases)
        - For example, create a helper function called toLower; the purpose of this function is to take in a string and return the lowercase of that string. The idea is to use this function on strings during case insensitive comparisons so that whichever case they may be we can tell if they are the same word or not.

- If both the listener name and the song name are found in the arrays, then the function should return the number of plays of that song listened by the listener.
- The function should return the following values depending on cases:
    - Return the number of plays if both listener and song name are found
    - Return -1 if the listener name is found but the song name is not found; this also applies if the song file is empty.
    - Return -2 if the listener name is not found but the song name is found; this also applies if the listener file is empty.
    - Return -3 if the listener and the song name both are not found
- The function does not print anything to the screen.

Sample code to generate `songs` and `listeners` array (this is to set up the arrays not the actual function call itself).

Songs2.txt	`Goodbye Yellow Brick Road,Elton John,Pop`
	`Turn Back the Hands of Time,Tyrone Davis,Rhythm and Blues`
	`I Want to Hold Your Hand,The Beatles,Rock`

Setting the values in `songs`	`// You can set values in songs using the readSongs function you developed in Problem 2 or just by using setters` `Song songs[50];`

	`int numSongsStored = readSongs("songs2.txt", songs, 0, 50);` `cout << numSongsStored << endl;`
Printing the values in `songs`	`printAllSongs(songs, numSongsStored);`

`//Expected Output`	`Here is a list of songs`
	`Goodbye Yellow Brick Road is by Elton John`
	`Turn Back the Hands of Time is by Tyrone Davis`
	`I Want to Hold Your Hand is by The Beatles`

playlist.txt	`Al,0,0,1`
	`John,5,0,2`
	`Sleve,1,0,1`
	`Onson,2,1,0`

Setting the values in `listeners`	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`numListenersStored =`
	`readListenerInfo("playlist.txt", listener,`
	`numListenersStored, lisArrSize, 50);`

Printing the values in `listeners`	`// Code to print the values`
	`for(int i = 0; i <numListenersStored; i++){`
	
	`cout<<listener[i].getListenerName()<<endl;`
	`for (int j = 0; j<numSongsStored; j++)`
	
	`cout<<listener[i].getPlayCountAt(j)<<endl;`
	
	`}`

`// Expected Output`	`Al`
	`0`
	`0`

```
1
John
5
0
2
Sleve
1
0
1
Onson
2
1
0
```

Example 1: Both the listener name and song name exists, and the number of listens is non-zero

Function call	`Song songs[50];`
	`int numSongsStored = readSongs("songs2.txt", songs, 0, 50);`
	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`numListenersStored = readListenerInfo("listenerInfo.txt", listener, numListenersStored, listenerArrSize, 50);`
	`cout<<getSongPlayCount("John", "Goodbye Yellow Brick Road", listener, songs, numListenersStored, numSongsStored)<<endl;`

Return value	`5`

Example 2: The listener name does not exist, it returns - 3

Function call	`Song songs[50];`
	`int numSongsStored = readSongs("songs2.txt", songs, 0, 50);`
	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`numListenersStored = readListenerInfo("listenerInfo.txt", listener, numListenersStored, listenerArrSize, 50);`

	`cout<<getSongPlayCount("John", "Master of Puppets", listener, songs, numListenersStored, numSongsStored)<<endl;`
Return value	`-1`

Example 3: The song name does not exist, it returns - 3

Function call	`Song songs[50];`
	`int numSongsStored = readSongs("songs2.txt", songs, 0, 50);`
	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`numListenersStored = readListenerInfo("listenerInfo.txt", listener, numListenersStored, listenerArrSize, 50);`
	`cout<<getSongPlayCount("Gene", "Goodbye Yellow Brick Road", listener, songs, numListenersStored, numSongsStored)<<endl;`

Return value	`-2`

Example 4: The listener name and song name do not exist

Function call	`Song songs[50];`
	`int numSongsStored = readSongs("songs2.txt", songs, 0, 50);`
	`int numListenersStored = 0;`
	`int listenerArrSize = 50;`
	`Listener listener[listenerArrSize];`
	`numListenersStored = readListenerInfo("listenerInfo.txt", listener, numListenersStored, listenerArrSize, 50);`
	`cout<<getSongPlayCount("Gene", "Master of Puppets", listener, songs, numListenersStored, numSongsStored)<<endl;`

Return value	`-3`

The zip submission should have five files for this problem: **Song.h**, **Song.cpp**, **Listener.h**, **Listener.cpp**, and a driver called **getSongPlayCountDriver.cpp**, with a `main()` function to test your member and **getSongPlayCount** function. For Coderunner, paste your Listener and Song classes (the contents of the four Listener and Song files mentioned above) and your **getSongPlayCount** function. ***Do not include the `main()` function.*** After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 9 (7 points): addListener

Write a standalone function **addListener** to add a listener to the `listeners` array. A new listener will have a play count of 0 for all songs. While adding a new listener to the array, you should check if a listener with the given name already exists. If the listener already exists, then the new listener should **not** be added to the array.

- The function should accept five arguments in this order:
    - **string** `listenerName`: name of the listener to be added to the array
    - **array** `listeners`: an array of listener objects
    - **Int** `numSongs`: the number of songs (this is a constant integer with value 50--the number of songs in our dataset).
    - **int** `numListenersStored`: number of listeners currently stored in the array
    - **int** `listenersArrSize`: the capacity of the `listeners` array. The default value for this constant integer is 100
- A new `listener` object will be added to the `listeners` array, provided that another listener with the same `listenerName` does not already exist (comparison should be **case insensitive** - cannot have Nick and nick in the database).
- If the `listenerName` is an empty string, it should not be added to the `listeners` array.
- The function should return an **integer** value depending on the situations below (check in order--so if listenerName is an empty string but the listener array is full, return -2):
    - Case 1: If `numListenersStored` is greater than or equal to the `listenerArrSize`, then the `listener` array is full and the function should **return -1**.
    - Case 2: If a listener with the same `listenerName` already exists, then do not add the listener to the array. The function should **return -2**.
    - Case 3: If `listenerName` is an empty string, then do not add the listener to the array. The function should **return -3**.
    - Case 4: Otherwise, add the listener object to the array and **return the new total number of listeners in the array**.

Example 1: The `listeners` array is full and we call the **addListener** function to add a new listener. The function cannot add another listener and returns -1.

Function call	`Listener listeners[1];`
	`int numSongs = 50;`
	`int listenersArrSize = 1;`
	
	`listeners[0].setListenerName("Ninja");`
	
	`// Add 5 songs listened to by the listener "Ninja"`
	`for(int i=0; i<5; i++) {`
	`  listeners[0].setPlayCountAt(i, 1);`
	`}`
	
	`int numListenersStored = 1;`
	
	`int val = addListener("Knuth", listeners, numSongs, numListenersStored, listenerArrSize);`

Return value	`-1`

Testing the data member `listenerName`	`// Code to print the values`
	`for(int i=0; i<numListenersStored; i++) {`
	`  cout<<listeners[i].getListenerName()<<endl;`
	`}`
	
	`// Expected Output:`
	`Ninja`

Example 2: The `listener` array is empty, and we call the **addListener** function to add a listener. The function creates and adds a **Listener** object to the `listeners` array, and returns the current number of listeners in the array (i.e. 1).

Function call	`Listener listeners[10];`
	`int listenersArrSize = 10;`
	`int numListenersStored = 0;`
	`int numSongs = 50;`
	
	`int val = addListener("Knuth", listeners, numSongs, numListenersStored, listenersArrSize);`

Function return value	`1`
	`// Code to print the listenerName and the number of listens for the first song`

	cout<<listeners[0].getListenerName()<<endl;
	cout<<listeners[0].getPlayCountAt(0)<<endl;
	
	// Expected Output
	Knuth
	0

Example 3: The `listeners` array is initially empty. We try to add a single listener twice. The function should add only one Listener object to the array.

Function call	Listener listeners[10];
	int listenersArrSize = 10;
	int numSongs = 5;
	
	listeners[0].setListenerName(“Knuth”);
	
	// Add listens for 5 songs for listener “Knuth”
	for(int i=0; i<5; i++) {
	    listeners[0].setPlayCountAt(i, i);
	}
	
	int numListenersStored = 1;
	
	cout<<addListener(“Knuth”, listeners, numSongs , numListenersStored, listenersArrSize)<<endl; // Prints -1

Function return value	-2

	// Code to print the listenerName and song three
	cout<<listeners[0].getListenerName()<<endl;
	cout<<listeners[0].getPlayCountAt(3)<<endl;
	cout<<listeners[1].getListenerName()<<endl;
	cout<<listeners[1].getPlayCountAt(3)<<endl;
	
	// Expected Output
	Knuth
	3
	“”
	0

Example 4: The listeners array has 2 listeners. We will add one new listener. The function adds this new listener and initializes all songs to 0. The function returns the new total number of listeners in the array.

Function call	`Listener listeners[10];`
	`int listenersArrSize = 10;`
	`int numSongs  = 5;`
	
	`listeners[0].setListenerName("Knuth");`
	`listeners[1].setListenerName("Richie");`
	
	`// Add play count for 5 songs`
	`for(int i=0; i<numSongs; i++) {`
	`    listeners[0].setPlayCountAt(i, i);`
	`    listeners[1].setPlayCountAt(i, 5-i);`
	`}`
	
	`int numListenersStored = 2;`
	
	`int val = addListener("Ninja", listeners,`
	`numSongs, numListenersStored,`
	`listenersArrSize)<<endl;`
	
	`cout<<"Total number of listeners in the array:`
	`"<<val<<endl;`

Output	`Total number of listeners in the array: 3`

	`// Code to print the play counts`
	`for(int i=0; i<numListenersStored; i++) {`
	`     cout<<listeners[i].getListenerName()<<" - ";`
	`     for(int j=0; j<numSongs; j++) {`
	`          cout<<listeners[i].getPlayCountAt(j)<<",`
	`";`
	`     }`
	`     cout<<endl;`
	`}`
	
	`// Expected Output`
	`Knuth - 0, 1, 2, 3, 4`
	`Richie - 5, 4, 3, 2, 1`
	`Ninja - 0, 0, 0, 0, 0`

The zip submission should have three files for this problem: **Listener.h**, **Listener.cpp** and a driver program called **addListenerDriver.cpp**, with your **addListener()** function and a **main()** function to test your **addListener()** function.

For **Coderunner**, paste your **Listener class** (both the header and implementation)**, and your addListener function**, not the entire program. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 10 (10 points): getListenerStats

Write a standalone function that, given a listener's name, prints the number of *unique* songs that listener has listened to, and the listener's average number of listens per song listened to (hint: remember the getSongPlayCount method from question 8). Note that if a listener has not listened to a song, those listens (0) should not be included in the average calculation.

- Your function **MUST** be named **getListenerStats**.
- Your function should take 4 parameters:
    - **string** `listenerName`: the name of the listener whose stats we want
    - **array of Listener objects**: `listeners`
    - **int** `numListenersStored`: number of listeners currently stored in the `listeners` array
    - **int** `numSongs`: the number of songs currently stored in the playCount array
- The function should perform the following actions depending on cases:
    - If the listener is found:
        - Print the results in the following format:
            - `<name> listened to <number> songs.`
            - `<name>'s average number of listens was <averagenumListens>`
        - Return 1
        - Note that the average number of listens should be rounded to 2 decimal places using the `setprecision()` function.
    - If the listener didn't listen to any songs:
        - Print the result in the following format:
            - `<name> has not listened to any songs.`
        - Return 0
    - If the listenerName is not found.
        - Print the result in the following format:
            - `<name> does not exist.`
        - Return -3

Sample code to generate `listeners` array.

Setting the values in `listeners`	`//Creating 3 listeners` `Listener listeners[3];`

```
//Setting listenerName and num listens for
Listener1
listeners[0].setListenerName("Listener1");
listeners[0].setPlayCountAt(0,1);
listeners[0].setPlayCountAt(1,4);
listeners[0].setPlayCountAt(2,2);

//Setting listenerName and num listens for
Listener2
listeners[1].setListenerName("Listener2");
listeners[1].setPlayCountAt(0,0);
listeners[1].setPlayCountAt(1,5);
listeners[1].setPlayCountAt(2,3);

//Setting listenerName and num listens for
Listener3
listeners[2].setListenerName("Listener3");
listeners[2].setPlayCountAt(0,0);
listeners[2].setPlayCountAt(1,0);
listeners[2].setPlayCountAt(2,0);
```

Example 1: The `listenerName` exists, and the listener has listened to at least 1 song

Function call	`getListenerStats("Listener2", listeners, 3, 3);`
Print	`Listener2 listened to 2 songs.` `Listener2's average number of listens was 4.00`
Return value	`1`

Example 2: The listener hasn't listened to any songs

Function call	`getListenerStats("Listener3", listeners, 3, 3);`
Print	`Listener3 has not listened to any songs.`
Return value	`0`

Example 3: The `listenerName` does not exist, it returns - 3

Function call	`getListenerStats("Listener4", listeners, 3, 3);`
Print	`Listener4 does not exist.`

Return value	`-3`

The zip submission should have three files for this problem: **Listener.h**, **Listener.cpp** and a driver program called **getListenerStatsDriver.cpp**, with your **getListenerStats()** function and a **main()** function to test your **getListenerStats()** function.

For **Coderunner**, paste your **Listener class** (both the header and implementation)**, and your getListenerStats function**, not the entire program. After developing in VSCode, this function will be one of the functions you include at the top of **project2.cpp** (for problem 11).

Problem 11 (20 points): Put them together

Now combine your `Song` class, `Listener` class, and functions you wrote so far. Create a file called **project2.cpp**. In this file write a program that gives the user a menu with the following options:

1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get song listens by a listener
8. Get listener statistics
9. Quit

NOTE: The *Song* class definition should be organized in *Song.h* and *Song.cpp*. Similarly, the *Listener* class definition should be organized in *Listener.h* and *Listener.cpp*. All the other functions used for Problems in Homework7/Project2 will go in *project2.cpp*.

For this problem (11), you need to submit the entire program *project2.cpp* (including *Song* class and *Listener* class) in the answer box for **Questions 11 and 12 on CodeRunner.**

The menu will run on a loop, continually offering the user options until they opt to quit. You should make use of the functions you wrote previously, call them, and process the values they return.

Size of arrays

In your driver function, you must declare your arrays with the appropriate size. The capacity of the `songs` array is 50. The capacity of the `listeners` array is 100.

Option 1: Read songs from file

- Prompt the user for a file name.
    - `Enter a song file name:`
- Pass the filename and other required arguments to your `readSongs` function.
- Print the total number of songs in the database in the following format:
    - `Total songs in the database: <numberOfSongs>`
- If the function returns -1, then print the following message:
    - `No songs saved to the database.`
- If the function returns -2, print
    - `Database is already full. No songs were added.`
- If the function returns a value equal to the size of the array of Songs print the following message (as well as the total number of songs message above):
    - `Database is full. Some songs may have not been added.`

Option 2: Print all songs

- Call your `printAllSongs` function with the required arguments.

Option 3: Song-count by genre

- Prompt the user for the `genre`.
    - `Enter the genre:`
- Read and pass the `genre` name to your `countGenre` function along with the other required arguments.
- Print the total number of songs for that category, in the database, in the following format:
    - `Total <genre> songs in the database: <count>`
    - ***Example***`: Total Rock songs in the database: 8`

Option 4: FrequentGenreSongs

- Call your `frequentGenreSongs` function, with all its required arguments.
- The terminal should print the total number of songs that belong to the most common genre, in the following format:
    - `Number of songs in most common Genre: <count>`
    - ***Example:*** `Number of songs in most common Genre: 8`

Option 5: Add listener

- Prompt the user for a listener name.
    - `Enter a listener name:`
- Pass the listener name, and other required arguments to your `addListener` function.
- If the `listeners` array is full (function returns -1), then print the following message
    - `Database is already full. Listener cannot be added.`
- If the listener name already exists (function returns -2), skip adding the listener and print the following message:
    - `Listener already exists.`

- If the listenerName is an empty string (function returns -3), skip adding the listener and print the following message:
    - `The listenerName is empty.`
- If the listener is added successfully, print the following message:
    - `Welcome, <listenerName>!`

Option 6: Read listener info from file

- Prompt the listener for a file name.
    - `Enter the listener info file name:`
- Pass the file name, and other required parameters to your `readListenerInfo` function.
- Print the total number of listeners in the database in the following format post insertion:
    - `Total listeners in the database: <numberOfListeners>`
- If the function returns -1, then print the following message:
    - `Nothing saved to the database.`
- If the function returns -2, print
    - `Database is already full. Nothing was added.`
- If the function returns a value equal to the size of the array of `listeners` print the following message (as well as the total listeners in database message above):
    - `Database is full. Some listeners may have not been added.`

Option 7: Get Song Play Count

- Prompt the user for a listener name.
    - `Enter a listener name:`
- Prompt the user for a song name.
    - `Enter a song name:`

Pass the listener name, song name, and other required parameters to your `getSongPlayCount` function.

- Print the result in the following format:
    - `<name> has listened to <songName> <numberOfListens> times.`
- If the function returned -1, print the result in the following format:
    - `<songName> does not exist.`
- If the function returned -2, print the result in the following format:
    - `<name> does not exist.`
- If the function returned -3, print the result in the following format:
    - `<name> and <songName> do not exist.`

Option 8: Get listener statistics

- Prompt the user for a listener name.
    - `Enter a listener name:`
- Pass the listenerName and other required parameters to your `getListenerStats` function.
- The terminal should display the output as printed by `getListenerStats`.

Option 9: Quit

- Print `Good bye!` and then stop the program.

Invalid input

If the user input is not the above values print `Invalid input.`

Below is an example of running the `project2` program:

```
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
1
Enter a song file name:
badFile.txt
No songs saved to the database.
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
1
Enter a song file name:
songs20.txt
Total songs in the database: 20
Select a numerical option:
======Main Menu=====
```

```
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
5
Enter a listener name:
Eve
Welcome, Eve!
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
5
Enter a listener name:
Eve
Listener already exists.
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
6
```

```
Enter the listener info file name:
short_listenData.txt
Total listeners in the database: 5
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
7
Enter a listener name:
Victoria
Enter a song name:
Call Me
Victoria has listened to Call Me 3 times.
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
7
Enter a listener name:
Victoria
Enter a song name:
Hello
Hello does not exist.
Select a numerical option:
======Main Menu=====
1. Read songs from file
```

```
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
8
Enter a listener name:
Eunice
Eunice does not exist.
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
8
Enter a listener name:
Eve
Eve has not listened to any songs.
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
11
Invalid input.
```

```
Select a numerical option:
======Main Menu=====
1. Read songs from file
2. Print all songs
3. Song-count by genre
4. Songs from most common genre
5. Add listener
6. Read listens from file
7. Get number of listens by a listener
8. Get listener statistics
9. Quit
9
Good bye!
```

Extra-credit: smartPlaylist (8 points)

The standalone function `smartPlaylist` will create a playlist of songs a listener might enjoy, based on the history of another listener who has listened to similar songs.

- Your function **MUST** be named **smartPlaylist**
- Your function should take 5 parameters in the following order:
    - **string**: `listenerName`
    - **string**: a `genre` to recommend
    - **Array of Listener objects**: `listeners`
    - **Array of Song objects**: `songs`
    - **int**: number of listeners currently stored in the `listeners` array
    - **int**: number of songs currently stored the `songs` array
- Your function should not return anything.
- Your function should find the listener with the given listener name, and print some song recommendations to the screen. (Details on how to recommend songs are given below.)
- The name search should be case insensitive. For example, “Ben, “ben” and “BEN” are all the same listener.
- If the listener name is not found, it should print the following message :

```
<listenerName> does not exist.
```

- If there are no songs to recommend for the listener, print the following:

```
    There are no recommendations for <listenerName> at
present.
```

- If there is at least one song to recommend for a certain listener, print the following information for **at most five** songs.

```
Here is the playlist:
Song: <song_name_1>, Artist: <artist_1>
Song: <song_name_2>, Artist: <artist_2>
…
…
Song: <song_name_5>, Artist: <artist_5>
```

For Coderunner, paste your **Listener** and **Song** classes and your **smartPlaylist** function.

How to find songs to recommend?
The recommendations for a given listener will be based on the other listener who is most similar to that listener. To generate recommendations, for example, for a listener named Ben:

1. Find the most similar listener to Ben. Let's say we found Claire to be most similar.
2. Recommend to Ben the first 5 songs in the database and in the specified genre that Claire has listened to, but that Ben has not yet listened to.
3. If there are fewer than 5 songs to recommend, recommend as many as possible. Ben will be presented with between 0 and 5 recommendations.

In order to compare two listeners and calculate their similarity, we will be looking at the play count for all the songs for **both** listeners, and calculating the similarity in their listening history. (Note we are matching songs by their index in the listener's playCount array--so we will compare person1.getPlayCountAt(1) to person2.getPlayCountAt(1).) As our similarity metric is based on the dot product of the playCount array, more similar listeners will have larger *similarity values*. Therefore, when Ben is compared to all other listeners in the database, the listener whose similarity score with Ben is *largest* will be the most similar listener (Claire).

Note: In the event of a tie between two listeners for being the most similar to the listener you are making recommendations for, make recommendations using the listener with the *higher* index within the listeners array.

The similarity metric you should use is the **dot product**. The **dot product** is calculated by summing the product of the corresponding elements in two playCount arrays from two listeners. Follow the example below.

Let *a* represent Ben's listening history, and *b* represent Claire's listening history.

a_i is the number of times Ben has listened to song *i*, and b_i is the number of times Claire has listened to it. *n* is the total number of songs.

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

Example 1 : Calculating the dot product

$john's\ listening\ history$: $[0,\ 1,\ 3,\ 5]$
$claire's\ listening\ history$: $[3,\ 0,\ 5,\ 0]$

$$Dot\ Product\ =\ (0\ *\ 3)\ +\ (1\ *\ 0)\ +\ (3\ *\ 5)\ +\ \ (5\ *\ 0)$$
$$Dot\ Product\ =\ 15$$

Example 2: listeners with very different histories will get a low dot product.

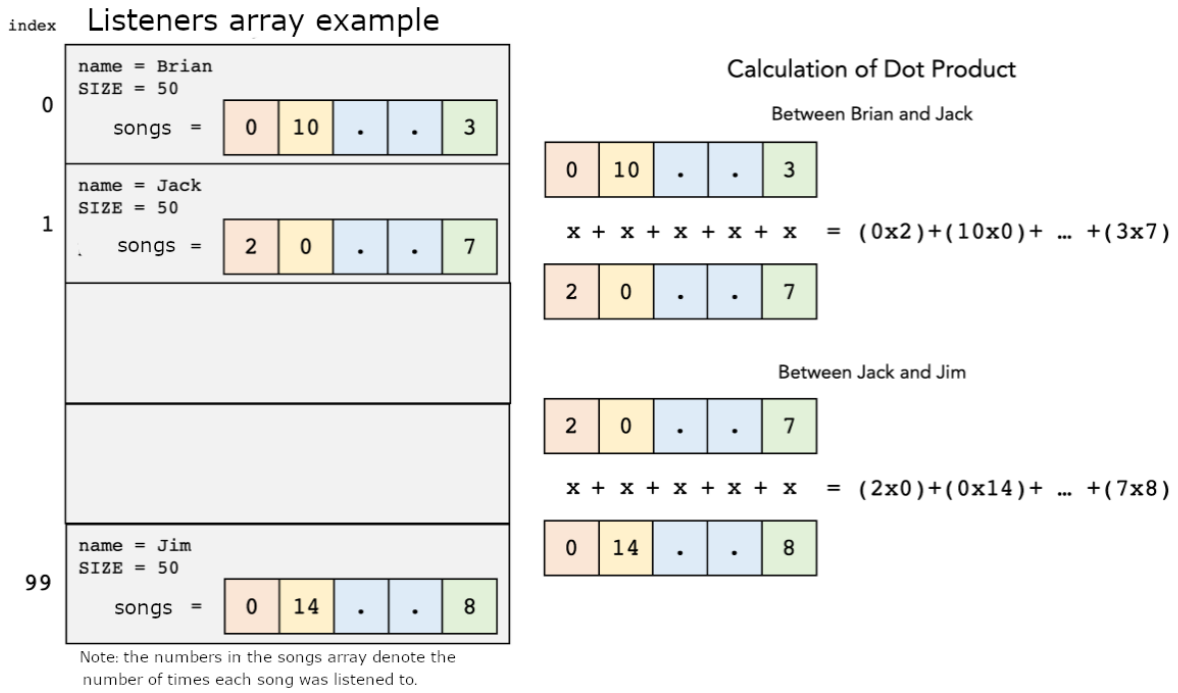
$john's\ listening\ history$: $[5,\ 1,\ 0,\ 0,\ 5]$
$david's\ listening\ history$: $[1,\ 5,\ 0,\ 5,\ 1]$

$$Dot\ Product\ =\ (5\ *\ 1)\ +\ (1\ *\ 5)\ +\ (0\ *\ 0)\ +\ \ (0\ *\ 5)\ +\ (5\ *\ 1)$$
$$Dot\ Product\ =\ 15$$

Example 3: Two listeners with very similar histories will get a higher dot product.

$john's\ listening\ history$: $[5,\ 0,\ 5,\ 3]$
$claire's\ \ listening\ history$: $[5,\ 0,\ 4,\ 2]$

$$Dot\ Product\ =\ (5\ *\ 5)\ +\ (0\ *\ 0)\ +\ (5\ *\ 4)\ +\ \ (3\ *\ 2)$$
$$Dot\ Product\ =\ 25\ +\ 20\ +\ 6\ =\ 51$$



For example
Let's say we're generating a playlist for John, in the genre Rock. Here are the songs:

```
She Loves You,The Beatles,Rock
I Want to Hold Your Hand,The Beatles,Rock
I Feel Fine,The Beatles,Rock
Ticket to Ride,The Beatles,Rock
```

```
Liz:   [5, 1, 5, 3]
John:  [5, 0, 3, 0]
David: [4, 1, 0, 5]
```

To generate recommendations for John:

1. **find the most similar listener**
   John has a dot product of 40 with Liz, and a dot product of 20 with David, so John is more similar to Liz. Thus, our song recommendations will be based on Liz's listening history.

2. **Find [at most] 5 songs Liz (the most similar listener) has listened to that John has not yet listened to.**
   We look at Liz's history to find songs that she has listened to that John has not:
   - Liz has listened to `She Loves You`, but John has already listened to this song.

- Liz has listened to `I Want To Hold Your Hand`. John hasn't listened to that song yet, so we add it to the playlist.
- Liz has listened to `I Feel Fine`, but John has already listened to this song.
- Liz has listened to `Ticket to Ride`. John has not yet listened to that song, so we add it to the playlist.
- There are no more songs that Liz has listened to, so we're done. Our final playlist will be:

```
Title: I Want to Hold Your Hand, Artist: The Beatles
Title: Ticket to Ride, Artist: The Beatles
```

Note that these are listed in the order they appear in the database.

Set-up for the examples:

songs.txt	`She Loves You,The Beatles,Rock`
	`I Want to Hold Your Hand,The Beatles,Rock`
	`I Feel Fine,The Beatles,Rock`
	`Ticket to Ride,The Beatles,Rock`

listensFile.txt	`Liz,5,1,5,3`
	`John,5,0,3,0`
	`David,4,1,0,5`

Set-up	`//Creating arrays`
	`Song songs[50];`
	`Listener listeners[100];`
	
	`int numSongsStored = 0;`
	`int numListenersStored = 0;`
	
	`//Setting song information for song array`
	`numSongsStored = readSongs("songs.txt", songs, 0, 50);`
	
	`//Setting listenerName and num listens for listeners array`
	`numListenersStored = readListens("listensFile.txt", listeners, 0, 100, 50);`

Example 1: There are songs to recommend

Function call	`smartPlaylist("John", "Rock", listeners, songs, numListenersStored, numSongsStored);`

outputs	`Here is the playlist:`
	`Title: I Want to Hold Your Hand, Artist: The Beatles`
	`Title: Ticket to Ride, Artist: The Beatles`

Example 2: No songs to recommend

Function call	`smartPlaylist("Liz", "Rock", listeners, songs, numListenersStored, numSongsStored);`
outputs	`There are no recommendations for Liz at present`

The best matched listener for Liz is John, with dot product = 40. As Liz has listened to all the songs John has, no recommendations are provided.

5. Project 2 checklist

Here is a checklist for submitting the assignment:

1. Complete the code **Project 2 - Coderunner**
2. Submit one zip file to **Project 2 zip submission**. The zip file should be named **Project2_lastname.zip**. It should have the following 15 files:
    - Song.h (from hmwk7)
    - Song.cpp (from hmwk7)
    - SongDriver.cpp (from hmwk7)
    - readSongDriver.cpp (from hmwk7)
    - printAllSongsDriver.cpp (from hmwk7)
    - countGenreDriver.cpp
    - frequentGenreSongsDriver.cpp
    - Listener.h
    - Listener.cpp
    - listenerDriver.cpp
    - readListenerInfoDriver.cpp
    - getSongPlayCountDriver.cpp
    - addListenerDriver.cpp
    - getListenerStatsDriver.cpp
    - project2.cpp
    - smartPlaylist.cpp *(if you attempted the EC problem)*

6. Project 2 points summary

Criteria	**Pts**
CodeRunner	80
Grading Interview	20
If absent from recitation:	-30
Total	100
Extra Credit	+8